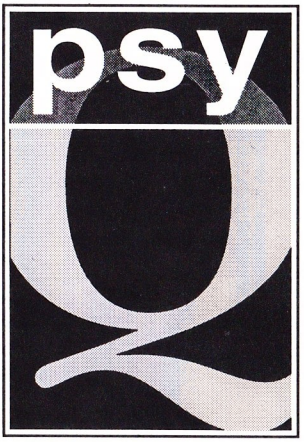
Programmer's Guide



Psy-Q ™ from Psygnosis Ltd

65816 Development System
for Nintendo Super-NES

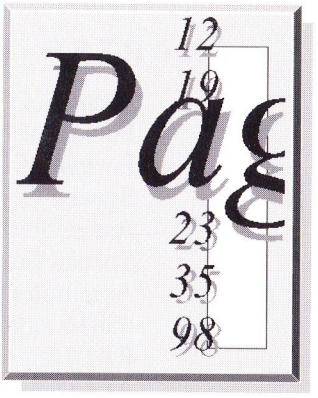
Information in this document is subject to change without notice.No part of this document may be transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express permission of Psygnosis Ltd.

© 1993, 1994 Psygnosis Ltd. All rights reserved.

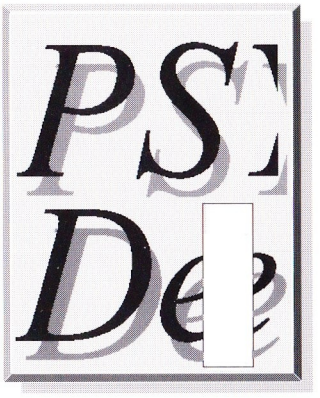
Psygnosis Ltd,
South Harrington Building,
Sefton Street,
Liverpool L34BQ

Document Number: BDL3005-3 (Nintendo)

Contents



Introduction



Welcome to **Psy-Q™** - the powerful development system linking PCs to Consoles and Games machines.

This version of **Psy-Q** features:

- ☐ **Fast 65816 Assembler**, capable of assembling at over a million lines of code per minute, and able to communicate directly with the target machine;
- ☐ **High Speed Linker and Librarian**, with extensive link-time options;
- ☐ **Powerful Source Level Debugger**, allowing the programmer to step, trace and set breakpoints directly in the source code.
- ☐ **Compact PC and Target machine adapters**, simple to fit, with no requirement to open or deface the target machine.

About Psy-Q

- **Psy-Q** has been developed by Psygnosis and S N Systems, with many years of experience of development software and developers' needs. Psy-Q represents the next generation of development systems, backed up by a commitment to continual enhancement, development and technical support.

- **Psy-Q** includes an industry-standard Assembler, Linker and Debugger. The Assembler is extremely fast, and fully compatible with other popular development systems. The Debugger offers an additional easy-to-use user interface, with full support for mouse and pop-down menus, and works in any text screen resolution.

- **Psy-Q** offers Source Level Debugging. This allows you to step, trace, set breakpoints, etc. in your original C or Assembler source code. The system automatically, and invisibly, handles multiple text files.

- **Psy-Q** provides a high-speed genuine SCSI parallel link between Host PC and target system, with a data transfer rate of up to 500 Kilobytes per second. The system supports up to 7 connected target devices, and cable lengths of over 6 metres.

- **Psy-Q**'s Assembler and Linker make full use of extended or expanded memory, on PC compatibles with more than 640K of RAM.

Psy-Q for Super-NES

The target interface is a compact steel-cased cartridge, that plugs into the cartridge slot of any ordinary, unmodified Super-NES console.

The adapter provides a through connector, to allow you to plug in and read production Nintendo cartridges, and to access hardware that may be on them.

Built-in adapter firmware provides diagnostics and self-test facilities. Also included are assorted functions for useful run-time control of the development environment, as well as extensive fileserver facilities, to allow the target to manipulate files on the host PC.

*Note that a standard Nintendo cartridge **must** be present for the unit to operate. This cartridge not only supplies the security chip, but can also provide facilities such as DSP chips and battery-backed RAM.*

Psy-Q Issue Information

Psy-Q development systems are available for a variety of platforms:

- Nintendo Super NES
- SEGA Mega Drive
- SEGA Mega-CD
- Commodore Amiga 1200

This **Psy-Q** development system, for the Nintendo Super-NES, is issued on a single diskette, which can be copied directly to a hard disk.

Contents of Issue Diskette:

ASM658.EXE	65816 Assembler
DBUG658.EXE	65816 Debugger
PSYLINK.EXE	Psy-Q Linker
PSYLIB.EXE	Psy-Q Librarian
RUN.EXE	Standalone Executable/Binary downloader
PSYMAKE.COM	Psy-Q Make Utility
PSYBIOS.COM	Psy-Q TSR BIOS extensions for PC host

Depending on the issue and version, the following files may also be included:

ASMSPC.EXE	SPC700 Assembler
MAKEFILE.MAK	Sample Makefile
PSYQ.CB	Brief Macros Source
PSYQ.CM	Compiled Brief Macros
xxxx.ICO	ICON Files, to aid installation of Psy-Q under Windows

Acknowledgements

Psy-Q The **Psy-Q** Development platform has been designed produced by S.N. Systems Limited, on behalf of Psygnosis Limited.

'Psy-Q' is a trademark of Psygnosis Ltd

Dos and Windows

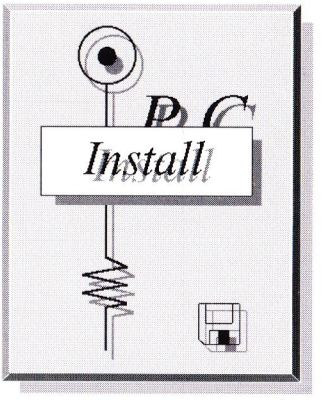
Microsoft, MS, MS-DOS are registered trademarks of Microsoft Corporation;
Windows is a trademark of Microsoft Corporation.

Brief Brief is a trademark of Borland International.

Nintendo Super-NES, Nintendo are all trademarks of Nintendo Co Ltd.

CHAPTER 1

PC Installation



The **Psy-Q** development system consists of the following physical components:

- ☐ PC Board
- ☐ Target Interface
- ☐ Connecting Cable
- ☐ PC driver and Bios extensions
- ☐ **Psy-Q** executable files

Installation is, therefore, a relatively straightforward procedure, and is described in this chapter under the following headings:

- Installation Check List
- PC Interface Installation
- PC software Installation
- **PSYBIOS**

See Chapter 14 - **Setting up the Target Machine** for the installation of the target adapter.

Installation check list

- ☐ Check the configuration of the Psy-Q PC board and install in the host PC - see page 5 for full installation details.
- ☐ Check configuration of the Psy-Q Target Adapter and install in the target console or machine - see page 207 for full installation details.
- ☐ Connect the supplied cable from the PC to the target machine.
- ☐ Load the PC board driver by typing, typically:

  PSYBIOS /a308 /d7 /i15

  See page 9 for full details of **PSYBIOS.COM**.
- ☐ Copy the runtime Psy-Q executable files to a directory on your PC - see the Issue List, in the Introduction, for the programs supplied with this version of Psy-Q.
- ☐ Switch on the target console, and insert a standard Nintendo cartridge.
- ☐ Run the program **RUN.EXE**, without parameters, to verify the link to the target adapter - see page 19 for details of the RUN downloader program.
- ☐ If **RUN** correctly identifies the target, the Psy-Q system is now ready to be used, to assemble, download and debug programs.

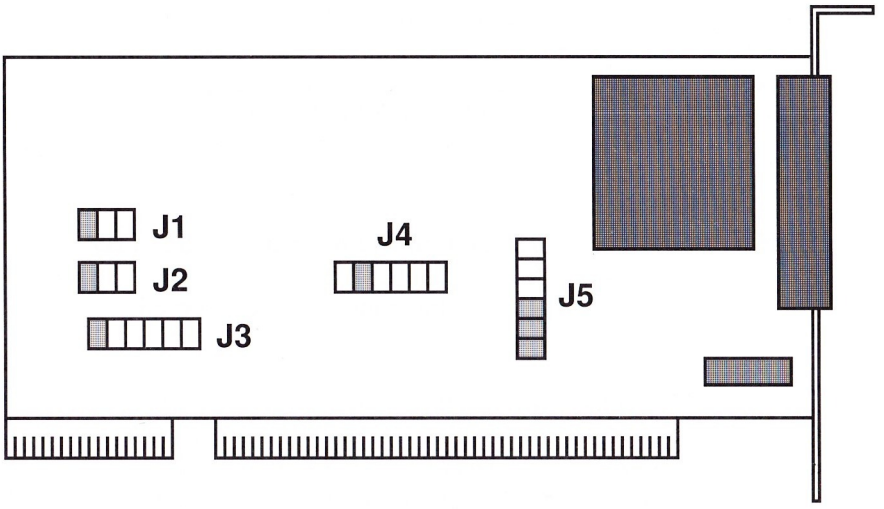
Note that, if the target interface adapter is configured with an ID other than zero, the target ID must be included on the command line of all development tools.

Installing the PC Interface

The **Psy-Q** PC Interface board should be fitted in to an empty 16 bit slot in the host PC. The host must be an IBM PC-AT or compatible, running under MSDOS 3.1 or better.

If no 16 bit slot is available, the board will also fit into an 8 bit slot. However, this may cause some degradation in speed.

Prior to fitting, the 5 sets of jumpers on the board should be checked and configured as required. It is likely, however, that the factory setting will suffice. The meaning of the jumpers is given below.



The **Psy-Q** PC Interface Board.

***CAUTION**: This board is sensitive to static electricity; hold by the metal support bracket when handling it.*

On-board Jumpers

J1 Select DRQ channel
J2 Select DACK channel

For J1 and J2, left to right jumper positions represent channels 7, 6, 5. Factory setting is 7, although Adaptec SCSI defaults to 5. Both J1 and J2 must be set to the same channel.

J3 Selects IRQ number.
From left to right, these are:
15,12,11,10,7,5.
Factory setting is 15

J4 Selects the base port address.
From left to right, these are:
300, 308, 310, 318, 380, 388, 390, 398 (hex).
Factory setting is 308.

J5 The bottom 3 jumpers are the SCSI ID; factory setting is 7.
The top three jumpers are reserved.

*The default settings have been chosen so that the possibility of contention with other internal boards is minimised. Nevertheless, care should be taken that settings on the **Psy-Q** board do not conflict with any other card in the system.*

Installing the PC Software

The Psy-Q issue diskette contains programs to perform the following functions:

- The Assembler
- The Linker
- The Debugger
- PC Driver
- Other target-specific Bios Extensions
- Windows accessories

Installing Development Software

To install the Psy-Q development programs onto the host PC, carry out the following procedure:

- create a directory on the hard disk e.g. C:\PSYQ;

- copy the contents of the issue disk to the new directory;

- add the new directory to the PATH variable in the AUTOEXEC.BAT;

- add a line in the AUTOEXEC.BAT to automatically load PSYBIOS.COM;

- add a line in the AUTOEXEC.BAT to create an environment variable "**PSYQ**", specifying the directory for **Psy-Q** files, e.g.:

  SET PSYQ = C:\PSYQ

Running from Windows

The Psy-Q Development system can readily be launched from within Windows, by performing the following:

- create a new ***program group*** using the NEW option in the FILE menu of PROGRAM MANAGER;
- create ***.PIF*** files, using the PIF Editor, to allow programs to run efficiently in a DOS window;
- create new ***program items***, again using the NEW option, for each Psy-Q facility to be run under Window;
- use CHANGE ICON... and the BROWSE facility to select an icon from the .ICO files, supplied with the issue disk.

PSYBIOS.COM

Description **PSYBIOS.COM** is a TSR program, which acts as a driver for the Psy-Q interface board, installed in the host PC.

Syntax **PSYBIOS** [*options*]

where each *option* is preceded by a forward slash (/) and separated by spaces.

Options

/a	*card address*	Set card address: 300, 308, 310, 318, 380, 388, 390, 398
/b	*size*	Specify file transfer buffer size: 2 to 32 (in kilobytes)
/d	*channel*	Specify DMA channel: 5, 6, 7; 0 = off
/i	*intnum*	Specify IRQ number: 5, 7, 10, 11, 12, 15; 0 = off
/s	*id*	Specify SCSI id: 0 to 7
/8		Run in 8 bit slot mode

Remarks

- Normally, PSYBIOS.COM is loaded in the AUTOEXEC.BAT; it can safely be loaded high to free conventional memory.
- If PSYBIOS is run again, with no options, the current image will be removed from memory. This is useful if you wish to change the options without rebooting the PC.
- If the DMA number is not specified, the BIOS will work without DMA; however, it will be slower.
- The BIOS can drive the interface in 8 bit mode; however, this is the slowest mode of operating the interface.
- The buffer size option (**/b**) sets the size of the buffer used when the target machine accesses files on PC. A larger buffer will increase the speed of these accesses; however, more PC memory will be consumed.

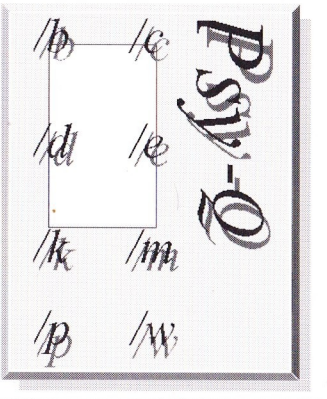
Examples

PSYBIOS /a308 /d7 /i15

Start the driver, using card address 308, DMA channel 7, interrupt vector 15; these are typical settings.

CHAPTER 2

The ASM658 Assembler



The ASM658 assembler is the backbone of the powerful Psy-Q development system; it can assemble 65816 source code at over 1 million lines per minute. Executable image or binary object code can be downloaded by the assembler itself, to run in the target machine immediately, or later, by the RUN utility.

This chapter discusses how to run an assembly session, under the following headings:

- Command Line Syntax
- Running with Brief
- Assembler and Target Errors
- RUN.EXE - download utility

Command Line Syntax

During the normal development cycle, the ASM658 Assembler may be:

- run in stand alone mode;
- launched from an editing environment, such as Brief - see later in this chapter;
- invoked as part of the Make utility - see page 193.

When the Assembler is run independently, the command line takes the following form; each component of the line is then described:

Syntax

ASM658 */switchlist source,object,symbols,listings,tempdata*

or

ASM658 *@commandfile*

If the first character on the command line is an @ sign, the string following it signifies a Psy-Q command file containing a list of Assembler commands.

Switches

The assembly is controlled by inclusion of a set of switches, each preceded by a forward slash (/). The **/o** switch introduces a string of assembler options; these can also be defined in the source code, using an **OPT** directive. Assembler options are described in detail on page 142, the available switches are listed below:

/c	Produce list of code in unsuccessful conditions
/d	Set Debug mode - if the object code is sent to the target machine, do not start it.
/e *n=x*	Assigns the value ***x*** to symbol ***n***.
/g	Non-global symbols will be output directly to the linker object file.
/i	While assembling, invoke the information window.
/j *pathname*	Nominate a search path for **INCLUDE** files.
/k	Permits the inclusion of pre-defined foreign conditionals, such as **IFND** - see also **MACROS**, page 111.
/l	Output a file for the Psylink Linker.
/m	Expand all macros encountered.
/o *options*	Specify Assembler options - see page 142 for a full description of the available options.
/p	Output pure binary object code, instead of an executable image in .CPE format - see also **RUN.EXE**, page 19.
/w	Output **EQUATE** statements to the Psylink file.
/z	Output line numbers to the Psylink file.
/zd	Generate source level Debug information.

Source The file containing the source code; if an extension is not specified, the default is **.658**. If this parameter is omitted, the Assembler outputs *help* in the form of a list of switches.

Object The file to which object code is written . If the object code is to be sent directly to the target machine, specify an filename of **T*n*:**, where ***n*** signifies the *SCSI* device number of the target. If this parameter is omitted, object code will not be produced.

Symbols The file to which symbol information is written, for use by the **DBUG658** Debugger.

Listings The file to contain listings generated by assembly.

Tempdata This parameter nominates a file to be placed on the RAM disk for faster access. If the name is omitted, the file will be called **ASM.TMP**; note that the temporary file is always deleted after assembly is complete.

Remarks

- If any of the above parameters are omitted, the dividing comma must still be included in the command line, unless it follows the last parameter.
- The Assembly run may be prematurely terminated by:

  - pressing **Control-C**;
  - pressing **Control-Break** (recognised more quickly because it does not require a DOS operation to spot it);

- pressing **Esc**, if the **/i** option has been specified to invoke the Info Box (this may be advisable with some versions of Brief that are erratic following **Control-C** or **Control-Break** - see page 17).

- The Assembler checks for an environment variable called **ASM658.** This can contain default options, switches and file specifications, in the form of a command line, including terminating commas for unspecified parameters. Defaults can be overridden in the runtime command line.

Examples

ASM658 /zd /o ae+,w- scode,t0:,scode.sym

This command will initiate the assembly of the source code contained in a file called **SCODE.658**, with the following active options:

- *source level debug information* to be generated;
- *automatic even* enabled;
- *warning messages* to be suppressed;
- the resultant *object code* to be transferred directly to the *target machine*, SCSI device 0;
- *symbol information* to be output to a file called **SCODE.SYM**;
- *assembly listing* to be suppressed.

ASM68K @game.pcf

will recognise the preceding @ sign and take its command line from a Psy-Q command file called GAME.PCF.

Running with Brief

Most programmers prefer to develop programs completely within a single, enabling environment. Future versions of Psy-Q will provide a self-contained superstructure with a built-in editor, tailored to the requirements of the assembly and debug sub-systems. For the time being, however, it is recommended that programmers seeking such facilities should use Borland's **Brief** Editor, which is already supported by Psy-Q.

Installation in Brief

- Copy the file PSYQ.CM, containing macros, into the \BRIEF\MACROS directory, or create it from source file PSYQ.CB;
- Set the BCxxx environment variable;
- Set the BFLAGS environment variable, with **-mpsyq** appended, to force the Psy-Q macro file to be loaded on start-up;
- On the next run of the Assembler or Brief, a set-up dialogue box is output, which allows defaults to be specified for output destination, options and switches.
- Because of problems in some versions of Brief, following a **Control-C** or **Control-Break**, it is recommended that the **/i** option is used on assembly. This will enable the Info Box, allowing **Esc** to be used for premature Assembly termination.

Assembly Errors

During the assembly process, errors may be generated as follows:

- by the assembler itself, as it encounters error conditions in the source code;
- by a failure during downloading of the object code.

Remarks

- **Appendix A** gives a full list of Assembler error messages.
- Errors during the download normally produce an error message, followed by an option to Retry, Bus Reset or Abort, such as:

Target not Available

Bus not Available

plus

Abort, **R**etry or **B**us Reset

RUN.EXE - program downloader

Description This program downloads runnable object code to the target machine.

Syntax **RUN** [*switches*] *file name* [[*switches*] *filename*] ..

where switches are preceded by a forward slash (/).

Remarks

- If RUN is executed without any runtime parameters, the program will simply attempt to communicate with the target adapter hardware. If successful, RUN displays the target identification; if the attempt fails, an appropriate error message is displayed.
- The file to be downloaded may contain:

- an executable image, output by the development system, in .CPE format. Up to 8 CPE files may be specified - if no extension is specified, .CPE is assumed;
- a raw binary image of a cartridge.

- For an executable file, execution will begin as indicated in the source code; for a binary ROM image, execution will begin as if the target machine had been reset with a cartridge in place.
- Multiple executable files may be specified. However, only the last executable address will apply - specified files are read from left to right.

- The following switches are available:

/h	Halt target - that is, download but do not run.
/t#	Use target SCSI id number #.
/u#	Use target unit number #.
/m20	Mode 20 for SNES ROM image.
/m21	Mode 21 for SNES ROM image.
/f+	Force load to SNES fast memory
/f-	Force load to SNES slow memory

Examples

RUN FORTRESS.BIN

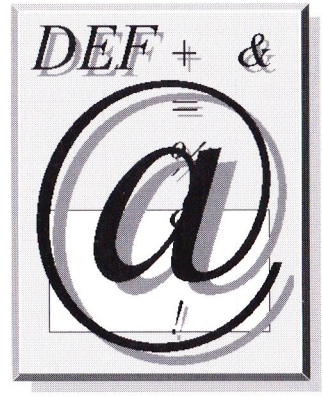
downloads the binary image FORTRESS.BIN to the target machine, at address #000000, and begins execution in a cartridge, as though the machine had been reset.

RUN SOLDIER.CPE

downloads the executable file SOLDIER.CPE to the target machine, and begins execution as indicated in the original source code.

CHAPTER 3

Syntax of Assembler Statements



In order to control the running of an Assembler, source code traditionally contains a number of additional statements and functions. These allow the programmer to direct the flow and operation of the Assembler as each section of code is analysed and translated into a machine-readable format. Normally, the format of Assembler statements will mirror the format of the host language, and **ASM658** follows this convention.

This chapter discusses the presentation and syntax of the **ASM658** statements, as follows:

- Format of Statements
- Format of Names and Labels
- Constants
- Functions
- Operators
- 65816 Addressing Mode
- RADIX
- ALIAS and DISABLE

Format of Statements

ASM658 statements are formatted as follows:

Name or Label *Directive* *Operand*

The following syntactical rules apply:

- Individual fields are delimited by spaces or tabs.
- Overlong lines can be split by adding an ampersand (&); the next line is then taken as a continuation.
- Lines with an equals (=) sign as the first character are considered to be the case options of a **CASE** statement - see *Flow Control*, page 81.
- Comment Lines:
  - comments normally follow the operand, and start with a semi-colon.
  - lines which consist of space or tab characters are treated as comments.
  - a complete line containing characters other than space or tabs is treated as a comment, if it starts with a semi-colon or asterisk.

Format of Names and Labels

Names and Labels consist of standard alpha-numeric symbols, including upper-case letters, lower-case letters and numeric digits:

A to **Z**, **a** to **z**, **0** to **9**

In addition, the following characters can occur:

Colon (**:**) Can used at the end of a name or label when defined, but not when referenced.

Question Mark (?), *Underscore* (_), *Dot* (.)
These three characters are often used to improve the overall readability

AT sign **@** Indicates the start of a local label - see page 121 onwards. Note that, by using the Assembler option ***/ln***, the local label symbol can be changed to a character other than @.

The following usage rules apply throughout:

- Numeric digits and Question Marks must not be the first character of a name.
- Labels normally start in column 1. However, if they start elsewhere, there must be no characters preceding the name, except space or tab, and the last character must be a colon.
- If a problem in interpretation is caused by the inclusion of a non-alphanumeric character in a Name or Label, that character can be replaced by a backslash, or the entire Name or Label surrounded by brackets.

Format of Constants

The **ASM658** Assembler supports the following constant types:

Character Constants

A character string enclosed in quote marks is a character constant and is evaluated as its ASCII value. Character constants may contain up to 4 characters, to give a 32 bit value. Thus:

"A" = 65
"AB" = (65*256)+66 = 16706
"ABC" = (65*65536) + (66*256)+67 = 4276803
"ABCD" = (65*16777216) + (66*65536) + (67*256)+68
= 1094861636

Integer Constants

Integer constants are normally evaluated as decimal, the default base, unless one of the following pertains:

- the **RADIX** directive changes the base - see page 37.
- **$**, as the first character of an integer, signifies a Hex number; **%** signifies a Binary number.
- If a character is preceded by a backslash and up arrow (\^), the corresponding control character is substituted.
- The **AN** Assembler option allows numbers to be defined as Intel and Zilog integers. That is, the number must start with a numeric character and end with one of:

  D for Decimal; **H** for Hexadecimal; **B** - Binary

Special Constants

The following pre-defined constants are available in **ASM658**:

_year	As a four digit number, e.g. 1995
_month	1 = January; 12 = December
_day	1 = 1st day of month
_weekday	0 = Sunday; 6 = Saturday
_hours	00 - 23
_minutes	00 - 59
_seconds	00 - 59
*	Contains the current value of the Location Counter.
@	Contains the actual PC value at which the current value will be stored - see below.
narg	Contains the number of parameters in the current macro argument - see page 110 for further details.
__rs	Contains the current value of RS Counter - see page 50 for further details.
_filename	A pre-defined string containing the name of the primary file undergoing assembly - that is, the file specified on the ASM658 command line.

Remarks

Time and Date Constants:

Time and Date constants are set to the start of assembly; they are not updated during the assembly process.

Example

```
RunTime     db      "\#_hours:\#_minutes:&
                    \#_seconds"
```

this expands to the form *hh:mm:ss*, as follows

```
RunTime     db      "21:08:49"
```

Note that this example uses the special macro parameter, \#, which is described on page 101.

Location Counter constants:

The current value of the program pointer can be used as a constant. To substitute the value of the location counter at the current position, an asterisk (*) is used:

```
            section Bss,g_bss
Firstbss    equ     *
```

Since * gives the address of the start of the line,

```
            org     $100
            dl      *,*,*
```

defines $100 three times.

An @, when used on its own as a constant, substitutes the value of the location counter, pointing to an address at which the current value will be stored.

```
            org     $100
            dl      @,@,@
```

defines $100,$104,$108.

Assembler Functions

ASM658 offers a large number functions to ease the programmer's task. Several are common to other assemblers - these are listed below, together with the page number for a more detailed explanation of their usage. In addition, there is a group of specialised functions, which are described on the following pages.

Name	*Action*	*Page*
def(**a**)	Returns true if **a** has been been defined	75
ref(**a**)	Returns true if **a** has been referenced	74
type(**a**)	Returns the data type of **a**	114
sqrt(**a**)	Returns the square root of **a**	
strlen(*text*)	Returns the length of string in characters	117
strcmp(*texta,textb*)	Returns true if strings match	118
instr([*start,*]*txa,txb*)	Locate substring **a** in string b	119
sect(**a**)	Returns the base address of section **a**	138
offset(**a**)	Returns the offset into section **a**	138
sectoff(**a**)	Equivalent to offset	138
group(**a**)	Returns the base address of group **a**	131
groupoff(**a**)	Returns the offset into group **a**	131

Special Functions

filesize("filename")

Returns the length of a specified file, or -1 if it does not exist.

groupsize(X)

Returns the current (***not final***) size of group X.

grouporg(X)

returns the ORG address of group X, or the group in which X is defined, if X is a symbol or section name.

groupend(X)

Returns the end address of group X.

sectend(X)

Returns the end address of section X.

sectsize(X)

Returns the current (***not final***) size of section X.

alignment(X)

Gives the alignment of previously defined symbol X. This value depends upon the base alignment of the section in which X is defined, as follows:

Longword aligned	- value in range 0 -3
Word aligned	- value in range 0 -1
Byte aligned	- value always 0

hi(X)

Gives the high byte of X; it operates the same as the **>X** function, and evaluates as (X>>8)&$FF.

lo(X)

Gives the low byte of X; it operates the same as the **<X** and evaluates as X&$FF.

__mx

Returns the current MX value.

Assembler Operators

The **ASM658** Assembler makes use of the following expression operators:

Symbol	*Type*	*Usage*	*Action*
( )	*Primary*	**(a)**	Brackets of Parenthesis
+	*Unary*	**+a**	**a** is positive
-	*Unary*	**-a**	**a** is negative (see Note [1])
=	*Binary*	**a=b**	Assign or equate **b** to **a**
+	*Binary*	**a+b**	Increment **a** by **b**
-	*Binary*	**a-b**	Decrement **a** by **b**
*	*Binary*	**a*b**	Multiply **a** by **b**
/	*Binary*	**a/b**	Divide **a** by **b**, giving the quotient
%	*Binary*	**a%b**	Divide **a** by **b**, giving the modulus
<<	*Binary*	**a<<b**	Shift **a** to the left, **b** times
>>	*Binary*	**a>>b**	Shift **a** to the right, **b** times
~	*Unary*	**~a**	Logical compliment or NOT **a**
&	*Binary*	**a&b**	**a** is logically ANDed by **b**
^	*Binary*	**a^b**	**a** is exclusively ORed by **b**
!	*Binary*	**a!b**	**a** is inclusively ORed by **b**
|	*Binary*	**a|b**	Acts the same as !
<>	*Binary*	**a<>b**	**a** is unequal to **b**
<	*Binary*	**a<b**	**a** is less than **b**
>	*Binary*	**a>b**	**a** is greater than **b**
<=	*Binary*	**a<=b**	**a** is less than or equals **b**
>=	*Binary*	**a>=b**	**a** is greater than or equals **b**

Note[1] Since the **ASM658** Assembler will evaluate 32-bit expressions, the negation bit is Bit 31. Therefore, $FFFFFF and $FFFFFFF are positive hex numbers; $FFFFFFFF is a negative number

Note[2] If a comparison evaluates as *true*, the result is returned as -1; if it evaluates as *false*, the result is returned is 0.

Hierarchy of Operators

Expressions in **ASM658** are evaluated using the following precedence rules:

- Parentheses form the primary level of hierarchy and force precedence - their contents are performed first;
- Without the the aid of parentheses, operators are performed in the order dictated by the hierarchy table;
- Operators with similar precedence are performed in the direction of their associativity - normally, from left to right, except unary operators.

Operator	*Direction*	*Description*
( )	←	Primary
+, -, ~	→	Unary
<<, >>	→	Shift
&, !, ^	→	Logical
*, /, %	→	Multiplicative
+, -	→	Additive
>, <, <=, >=	→	Relational
=, <>	→	Equality

65816 Addressing Modes

This section discusses some general topics pertaining to the 65816 Addressing Modes, as follows:

Immediate Addressing

<, > and ^ are additional prefix operators to follow #, which act as in the following examples:

```
lda #$12        ;load accumulator with immediate
                ;byte/word, depending on whether it's in 8
                ;or 16 bit mode

lda #<value     ;load accumulator with lower byte/word of
                ;value, depending on whether it's in 8 or
                ;16 bit mode

lda #>value     ;load accumulator with middle byte/word
                ;of value, depending on whether it's in 8 or
                ;16 bit mode

lda #^value     ;load accumulator with upper byte/word of
                ;value, depending on whether it's in 8 or
                ;16 bit mode
```

To summarise:

Operand	*Value in 8 bit mode*	*Result in 16 bit mode*
#<$12345678	$78	$5678
#>$12345678	$56	$3456
#^$12345678	$34	$1234

- If none of the prefix operators < > or ^ are used, the value specified will be loaded into the accumulator. If this value is out of range for the current accumulator size, an error will result, unless the truncate option (**opt t+**) has been specified.

Direct Page, Absolute, Long Absolute

Specifying one of the prefixes < | and > will force the Assembler to use the addressing mode indicated in the examples below:

```
lda  addr      ;load accumulator from address

lda  <addr     ;load accumulator from address in direct
               ;page

lda  |addr     ;load accumulator using absolute address

lda  >addr     ;load accumulator using long absolute
               ;address
```

- If address truncation (**opt ta+**) is in effect, the Assembler will truncate the operand address. Otherwise, an error will be generated if the operand address is outside the legal range for the mode specified. For instance,

  ```
  lda    <$123
  ```

  will result in an error.

- If one of the < | or > prefixes is not specified on the operand, the Assembler obeys the following rules to determine which addressing mode to use - see also page 89 on the ASSUME directive:

- If direct page optimisation is enabled (**opt od+**), the Assembler will attempt to determine whether **Direct Page Access** can be used, based on the current ASSUME value of the Direct Page Register. If Direct Page Addressing is not legal for the specified operation, no attempt to optimise to Direct Page Addressing will be made.

  If the Direct Page Register has no ASSUME value, or is ASSUMEd to ?, the Assembler will use Direct Page Addressing, if the operand is in the range 0-255 (decimal).

  If the Direct Page Register is ASSUMEd to a known value, the Assembler will use Direct Page Addressing if the operand address is from 0 to 255 (decimal) greater than this value, unless it is greater than 65535.

  If the Direct Page Register is ASSUMEd to a section name, the Assembler will use Direct Page Addressing if the operand is known to be in that section, at the point that the reference is encountered on pass 1.

  If the Direct Page Register is ASSUMEd to a group name, the Assembler will use Direct Page Addressing if the operand is known to be in a section in that group, at the point that the reference is encountered on pass 1.

- If Direct Page Addressing cannot be used, the Assembler will test to see whether **Long Absolute Addressing** must be used. If the data bank register has no ASSUME value, or is ASSUMEd to ?, the Assembler will use Long Absolute Addressing if the operand is 65536 (decimal) or greater.

If the Data Bank Register is ASSUMEd to a known value, the Assembler will use Long Absolute Addressing if the operand address is less than the value, or more than 65536 (decimal) greater.

If the Data Bank Register is ASSUMEd to a section name ,and the ORG address of that section can be found, the Assembler will use Long Absolute Addressing if the operand value is less than the ORG address, or more than 65536 (decimal) greater than the address.

If the Data Bank Register is ASSUMEd to a group name, and the ORG address of that group can be found, the Assembler will use Long Absolute Addressing if the operand value is less than the Org address, or more than 65536 (decimal) greater than the address.

- ☐ If it is not necessary to use Long Absolute Addressing, the Assembler will use **Absolute Addressing**.

RADIX

Description The **ASM658** Assembler defaults to a base of 10 for integers. This may be changed by preceding individual numbers by the characters % or $, to change the base for that integer to binary or hex . Alternatively, the RADIX directive can be used to change the default base.

Syntax **RADIX** *newbase*

Remarks

- Acceptable values for the new base are in the range of 2 to 16.
- Whatever the current default, the operand of the RADIX directive is evaluated to a decimal base.
- The AN assembler option (see page) will not be put into effect if the default RADIX is greater than 10, since the signifiers **B** and **D** are used as digits in hexadecimal notation.

Examples

```
radix   8
```

sets the default base to OCTAL.

ALIAS and DISABLE

Description These directives allow the programmer to avoid a conflict between the reserved system names of constants and functions, and the programmer's own symbols. Symbols can be renamed by the **ALIAS** directive and the original names **DISABLE**'d, rendering them usable by the programmer.

Syntax

newname **ALIAS** *name*
DISABLE *name*

Remarks

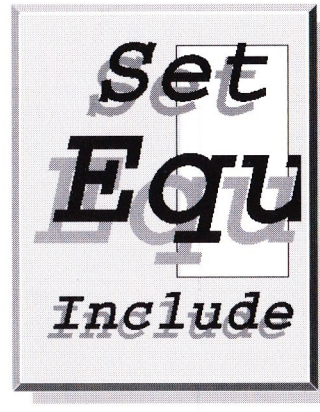
- Symbolic names currently known to the ASM658 Assembler may **ALIAS**ed and **DISABLE**d. However, these directives must not be used to disable ASM658 directives.

Examples

```
_Offset     alias     offset
            disable   offset
            ...
_Offset     dw        _Offset(Lab)
offset      dw        *-pointer
```

CHAPTER 4

General Assembler Directives



The **ASM658** assembler provides a variety of functions and directives to control assembly of the source code and its layout in the target machine.

This chapter documents the Assembler directives which allow the programmer to control the processes of assembly, grouped as follows:

- Assignment Directives
- Data Definition
- Controlling Program Execution
- Include Files
- Controlling Assembly
- Handling the Target Registers

Assignment Directives

The directives in this section are used to assign a value to a symbolic name. The value may a constant, variable or string.

- **EQU**
- **SET** (and =)
- **EQUS**
- **RB, RW, RL and RT**
- **RSSET**
- **RSRESET**

EQU

Description Assigns the result of the expression, as a **constant** value, to the preceding symbolic name.

Syntax *symbol name* **EQU** *expression*

See Also **SET, EQUS**

Remarks

- The **ASM658** Assembler allows the assigned expression to contain forward references. If an **EQU** cannot be evaluated as it is currently defined, the expression will be saved and substituted in any future references to the equate (see Note [1] below).
- It is possible to include an equate at assembly time, on the Assembler command line. This is useful for specifying major options of conditional assembly, such as *test mode* - see Assembler switches, page 13.
- Assigning a value to a symbol with **EQU** is absolute; an attempt at secondary assignment will produce an error. However, it is permissible to re-assign the ***current*** value to an existing symbol; typically, this occurs when subsidiary code redefines constants already used by the master segment.

Examples

```
Length      equ     4
Width       equ     8
Depth       equ     12
Volume      equ     Length*Width*Depth
DmaHigh     equ     $ffff8609
DmaMid      equ     DmaHigh+2
```

Note[1]

```
List        equ     Lastentry-Firstentry
```

if Firstentry, Lastentry not yet defined, then:

```
            dl      List+2
```

will be treated as

```
            dl      (Lastentry-Firstentry)+2
```

the equated expression is implicitly bracketed.

SET

Description Assigns the result of the expression, as a **variable**, to the preceding symbolic name.

Syntax

symbol name	**SET**	*expression*
symbol name	=	*expression*

See Also **EQU**

Remarks

- The **ASM658** Assembler does not allow the assigned expression in a **SET** directive to contain forward references. If a **SET** cannot be evaluated as it is currently defined, an error is generated.
- If the symbol itself is used before it is defined, **ASM658** generates a warning, and assigns the value determined by the preliminary pass of the Assembler.
- The symbol in a **SET** directive does not assume the type of the operand. It is, therefore, better suited to setting local values, such as in Macros, rather than in code with a relative start position, such as a **SECTION** construct, which may cause an error.

Examples

```
Loopcount   set     0
GrandTotal  =       SubTotalA+SubTotalB
xdim        set     Bsize<<SC
```

```
cbb         macro   string
lc          =       0
            rept    strlen(\string)
cc          substr  lc+1,lc+1,\string
            db      '\cc'^($A5+lc)
lc          =       lc+1
            endr
            endm
```

EQUS

Description Assigns a text or string variable to a symbol.

Syntax

symbol name	**EQUS**	*"text"*
symbol name	**EQUS**	*'text'*
symbol name	**EQUS**	*symbol name*

See Also **EQU, SET**

Remarks

- Textual operands are delimited by double or single quotes. If it is required to include a double quote in the text string, delimit with single quotes or two double quotes; similarly, to include a single quote in the text, delimit with double quotes or two single quotes - see examples below.

- If delimiters are omitted, the Assembler assumes the operand to be the *symbol name* of a previously defined string variable, the value of which is assigned to the new symbol name.

- Point brackets, { and }, are special delimiters used in Macros - see **MACRO** directive specification, page 101.

Symbols equated with the **EQUS** directive can appear at any point in the code, including as part of another text string. If there is the possibility of confusion with the surrounding text, a backslash (\) may be used before the symbol name, and, if necessary, after it, to ensure the expression is expanded correctly - see examples below.

Examples

```
Program     equs    "Psy-Q v 1.2"

Qtex        equs    "What's the score?"

            db      "Remember to assemble &
                    \_filename",0
```

```
z           equs    "123"
            ...
            dl      z+4
converts to
            dl      123+4
```

whereas the following expression needs backslashes to be expanded correctly:

```
            dl      number\z\a
converts to
            dl      number123a
```

```
SA          equs    'StartAddress'
            ...
            dl      \SA\4
converts to
            dl      StartAddress4
```

To include single quotes in a string delimited by single quotes, either change the delimiters to double quotes, or double-up the internal single quote. Similarly, this syntax applies to double quotes, as follows:

```
Sinquote    equs    'What''s the point?'
Sinquot2    equs    "What's the point?"

Doubquote   equs    "Say ""Hello"" and go"
Doubquote   equs    'Say "Hello" and go'
```

RB, RW, RL and RT

Description These directives assign the value of the **__RS** variable to the symbol, and advances the *r*s counter by the specified number of bytes.

Syntax

symbol name	**RB**	*count*
symbol name	**RW**	*count*
symbol name	**RL**	*count*
symbol name	**RT**	*count*

See Also **RSSET, RSRESET**

Remarks

- These directives, together with the following two associated directives, operate on or with the **__RS** variable, which contains the current offset.

- The directives RB, RW and RL will allocate one, two and four bytes respectively; the RT directive will allocate three bytes.

Examples

```
                rsreset

Icon_no         rb      2
Dropcode        rw      1
Actcode         rw      1
Actname         rb      10
Objpos          rl      1
Errcode         rt      2

Artlen          rb      0
```

After each of the first six RS equates, the **__RS** pointer is advanced; the values for each equate are as follows:

```
Icon_no         0       (set to zero by RSRESET)

Dropcode        2
Actcode         4
Actname         6
Objpos          16
Errcode         20

Artlen          26
```

The last **rb** does not advance the **__RS** pointer, since it is equivalent to:

```
artlen          equ     __RS
```

RSSET

Description Assigns the specified value to **__RS** variable.

Syntax **RSSET** *value*

See Also **RS, RSRESET**

Remarks

- This directive is normally used when the offsets are to start at a value other than zero.

Examples See the **RS** directives

RSRESET

Description Sets the **__RS** variable to *zero*.

Syntax **RSRESET** *[value]*

See Also **RS, RSSET**

Remarks

- Using this directive is the normal way to initialise the **__RS** counter at the start of a new data structure.
- The optional parameter is provided for compatibility with other assemblers; if present, **RSRESET** behaves like the **RESET** directive.

Examples See the **RS** directives

Data Definition

The directives in this section are used to define data and reserve space.

- **DB, DW, DL and DT**
- **DCB, DCW, DCL and DCT**
- **DS**
- **HEX**
- **DATA**
- **DATASIZE**
- **IEEE32**
- **IEEE64**

DB, DW, DL and DT

Description These directives evaluate the expressions in the operand field, and assigns the results to the preceding symbol, in the format specified by the second character of the directive. Argument expressions may be numeric values, strings or symbols.

Syntax

symbol name	**DB**	*expression,expression*
symbol name	**DW**	*expression,expression*
symbol name	**DL**	*expression,expression*
symbol name	**DT**	*expression,expression*

See Also **DCB, DCW, DCL, DCT**

Remarks

- Textual operands are delimited by double or single quotes. If it is required to include a double quote in the text string, delimit with single quotes or two double quotes; similarly, to include a single quote in the text, delimit with double quotes or two single quotes - see examples below. If delimiters are omitted, the Assembler expects the operand to be the *symbol name* of a previously-defined string variable, the value of which is assigned to the new symbol name.

- The DB, DW and DL directives operate to units of one, two and four bytes respectively; the DT directive uses units of three bytes.

Examples

```
Hexvals     dw      $80d,$a08,0,$80d,0
Coords      dw      -15,46
Pointers    dl      StartMarker,EndMarker
ErrorMes    db      "File Error",0
```

Notes

While some assemblers truncate a parameter that is out-of-range, **ASM658** flags an error; all of the following will produce errors:

```
db      257
db      -129
dw      66000
dw      -33000
```

DCB, DCW, DCL and DCT

Description These directives generate a block of memory, of specified *length*, containing the specified *value*.

Syntax

DCB *length,value*
DCW *length,value*
DCL *length,value*
DCT *length,value*

See Also **DB, DW, DL, DT**

Remarks

- The DCB, DCW and DCL directives generate memory blocks in units of one, two and four words respectively; the DCT directive generates units of three words.

Examples

```
dcb     256,$7F
```

generates 256 bytes containing $7F.

```
dcw     64,$FF
```

generates 64 words containing $FF.

DS

Description Allocates memory to the *symbol*, of the specified *length*, and initialises it to zero.

Syntax *symbol name* **DS** *length*

Remarks

- The **DS** directive reserves memory blocks to a unit of one byte.
- If the **DS** directive is used to reserve space in a Group/Section with the **BSS** attribute, the reserved area will ***not*** be initialised - see **Groups and Sections**, page 131.

Examples

```
List        ds      64
```

reserves an area 64 bytes long, and sets it to zero.

```
Buffer      ds      1024
```

reserves a 1k bytes area, and sets it to zero.

HEX

Description This directive takes a list of unsigned hex nibble pairs as an argument, which are concatenated to give bytes. It is intended as a quick way of inputting small hex expressions.

Syntax *symbol name* **HEX** *hexlist*

See Also **INCBIN**

Remarks Data stored as **HEX** is difficult to read, less memory-efficient and causes more work for the Assembler. Therefore, it is suggested that the HEX statement is used for comparatively minor data definitions only. To load larger quantities of data, it is recommended that the data is stored in a file, to be INCLUDEd as a binary file at runtime - see **Include Files**, page 72.

Examples

```
HexString   hex   100204FF0128
```

is another way of writing

```
HexString   db    $10,$02,$04,$FF,$01,$28
```

DATASIZE and DATA

Description Together, these directives allow the programmer to define values between 1 and 256 bytes long (8 to 2048 bits). The size of the **DATA** items must first be defined by a **DATASIZE** directive.

Syntax

DATASIZE *size*
DATA *value, value*

where *value* is a numeric string, in hex or decimal, optionally preceded by a minus sign.

See Also **IEEE32, IEEE64**

Remarks

- If a *value* specified in the **DATA** directive converts to a value greater than can be held in size specified by **DATASIZE**, the **ASM658** assembler flags an error.

Examples

```
datasize    8
...
data        $123456789ABCDEF0
data        -1,$FFFFFFFFFFFF
```

IEEE32 and IEEE64

Description These directives allow 32 and 64 bit floating point numbers to be defined in IEEE format.

Syntax

IEEE32 *fp.value*
IEEE64 *fp.value*

See Also **DATA, DATASIZE**

Examples

```
ieee32    1.23,34e10

ieee64    123456.7654321e-2
```

Controlling Program Execution

The directives in this section are used to alter the state of the program counter, and control the execution of the Assembler.

- **ORG**
- **CNOP**
- **OBJ**
- **OBJEND**

ORG

Description The **ORG** directive informs the Assembler of the location of the code in the target machine.

Syntax **ORG** *address[,parameter]*

where *address* is a previously-defined symbol, or a hex or decimal value, optionally preceded by a question mark (*?*) and followed by a (target-specific) numeric parameter.

See Also **OBJ, OBJEND, GROUP, SECTION**

Remarks

- If link file is output, the **ORG** directive must not be used - see **Groups and Sections**, page 131.
- If the program contains **SECTION**s, a single **ORG** is allowed, and it must precede all **SECTION** directives. If the program does not utilise the **SECTION** construct, it may contain multiple **ORG**'s.
- The **ORG** operand can be preceded by a question mark, to indicates the amount of RAM required by the program. However, the **ORG ?** function only works on machines with operating systems to allocate the memory; for instance, it will work on the Amiga but not the SNES or Sega Mega Drive.

Examples

```
                org     $100

Begin           lda     #%00000011
                sta     $4300
                lda     #$21
                sta     $4301
```

```
Program         equ     $4000
                ...
                org     Program
```

CNOP

Description Resets the program counter to a specified *offset* from the specified *size boundary*.

Syntax **CNOP** *offset,size boundary*

Remarks

- In code containing **SECTION**s, the **ASM658** Assembler does not allow the program counter to be reset to a size boundary greater than the alignment already set for that section. Therefore, a **CNOP** statement, with a size boundary of 2, is not allowed in a section that is byte-aligned.

Examples

```
                section.l   prime
Firstoff        =           512
Firstsize       =           2
                ...
                cnop        Firstoff,Firstsize
```

sets the program counter to 512 bytes above the next word boundary.

OBJ and OBJEND

Description **OBJ** forces the code following it to be assembled as if it were at the specified address, although it will still appear following on from the previous code. **OBJEND** terminates this process and returns to the **ORG**'d address value.

Syntax

OBJ *address*

OBJEND

See Also **ORG**

Remarks

- The **OBJ** - **OBJEND** construct is useful for code that must be assembled at one address (for instance, in a ROM cartridge), but will be run at a different address, after being copied there.
- Code blocks delimited by **OBJ** - **OBJEND** cannot be nested.

Examples

```
org     $100

dl      *
dl      *

obj     $200
```

```
dl      *
dl      *

objend

dl      *
dl      *
```

The above code will generate the following sequence of longwords, starting at address $100:

```
$100
$104
$200
$204
$110
$114
```

Include Files

The source code for most non-trivial programs is too large to be handled as a single file. It is normal for a program to be constructed of subsidiary files, which are called together during the assembly process.

The directives in this section are used to collect together the separate source files and control their usage; also discussed are operators to aid the control of code to be assembled from INCLUDEd files.

- **INCLUDE**
- **INCBIN**
- **DEF**
- **REF**

INCLUDE

Description Informs the **ASM658** assembler to draw in and process another source file, before resuming the processing of the current file.

Syntax

INCLUDE *filename*

where *filename* is the name of the source file to be processed, including drive and path identifiers - see **Note**[1]. The *filename* may be surrounded by quotes, but they will be ignored.

See Also **INCBIN**

Remarks

- Traditionally, there will be one main file of source code, which contains **INCLUDE**'s for all the other files.
- **INCLUDE**d files can be nested.
- The **/j** switch can be used to specify the search path for INCLUDEd files - see Assembler Options, page 142.

Examples A typical start to a program may be:

```
            section     short1
codestart   jmp         entrypoint

            db          _hours,_minutes
            db          _day,_month
            dw          _year
```

```
                include     vars1.658

                section     short2

                include     vars2.658

                section     code

                include     graph1.658
                include     graph2.658
                include     maths.658
                include     trees.658
                include     tactics.658

entrypoint      lda         #%00000011
                sta         $4300
                lda         #$21
                sta         $4301
                ...
```

Note[1]

Since a path name contains backslashes, the text in the operand of an **INCLUDE** statement may be confused with the usage of text previously defined by an **EQUS** directive. To avoid this, a second backslash may be used, or the backslash may be replaced by a forward slash (solidus).

Thus,

```
                include     d:\source\levels.asm
```

may be written as

```
                include     d:\\source\\levels.asm
```

or

```
                include     d:/source/levels.asm
```

INCBIN

Description Informs the **ASM658** Assembler to draw in and process binary data held on another source file, before resuming processing of the current file.

Syntax *symbol* **INCBIN** *filename*

- *filename* is the name of the source file to be processed, including drive and path identifiers - see **Note**[1]. Optionally, the *filename* may be surrounded by quotes, which will be ignored.
- *start, length* are optional values, allowing selected portions of the specified file to be included - see **Note** [2].

See Also **INCLUDE, HEX**

Remarks

- This directive allows quantities of binary data to be maintained in a separate file and pulled into the main program at assembly time; typically, such data might be character movement strings or location co-ordinates. The Assembler is passed no information concerning the type and layout of the incoming data. Therefore, labelling and modifying the incoming data are the responsibility of the programmer.

- The **/j** switch can be used to specify the search path for INCLUDEd files - see Assembler Options, page 142.

Examples

```
Charmove    incbin    "d:\source\charmove.asm"
```

Note[1]

Since a path name contains backslashes, the text in the operand of an **INCBIN** statement may be confused with the usage of text previously defined by an **EQUS** directive. To avoid this, a second backslash may be used, or the backslash may be replaced by a forward slash (solidus).

Thus,

```
            incbin    d:\source\theme.asm
```

may be written as

```
            incbin    d:\\source\\theme.asm
```

or

```
            incbin    d:/source/theme.asm
```

Note[2]

The nominated file may be accessed selectively, by specifying a position in the file, from which to start reading, and a length. Note that:

- if *start* is omitted, the INCBIN commences at the beginning of the file;
- if the *length* is omitted, the INCBIN continues to the end of the file;
- if both *start* and *length* are omitted, the entire file is INCBINed.

REF

Description **REF** is a special operator, to allow the programmer to determine which segments of code are to be INCLUDEd.

Syntax [~]**REF**(*symbol*)

The optional, preceding tilde is synonymous with ***NOT****.*

Remarks **REF** is *TRUE* if a reference has previously been encountered for the symbol in the brackets.

Examples

```
        if      ref(Links)
Links   lda     #%00000011

        ...
        rts
        endif
```

The *Links* routine will be assembled if a reference to it has already been encountered.

DEF

Description Like the REF operator, **DEF** is a special function. It allows the programmer to determine which segments of code have already been INCLUDEd.

Syntax **[~]DEF**(*symbol*)

*The optional, preceding tilde is synonymous with **NOT**.*

Remarks **DEF** is TRUE if the symbol in the brackets has previously been defined.

Examples

```
                if          ~def(load_addr)
load_addr       equ         $1000
exec_addr       equ         $1000
reloc_addr      equ         $80000-$300
                endc
```

The address equates will be assembled if *load_addr* has not already been defined.

Assembly Flow Control

The following directives give instructions to the ASM658 assembler, during the assembly process. They allow the programmer to select and repeat sections of code:

- **END**
- **IF**
- **ELSE**
- **ELSEIF**
- **ENDIF**
- **CASE**
- **ENDCASE**
- **REPT**
- **ENDR**
- **WHILE**
- **ENDW**
- **DO**
- **UNTIL**

END

Description The **END** directive informs the Assembler to cease its assembly of the source code.

Syntax **END** *[address]*

See Also **REGS**

Remarks

- The inclusion of this directive is mostly cosmetic, since the Assembler will cease processing when the input source code is exhausted.
- The optional parameter specifies an initial address for the program. See also the **REGS** statement, in the section - **Handling the Target Registers**, page 99.

Examples

```
startrel    lda     #%00000011
            ...
            jmp     progad
            end     startrel
```

IF, ELSE, ELSEIF, ENDIF, ENDC

Description These conditional directives allow the programmer to select code for assembly.

Syntax

IF *[~]expression*
ELSE
ELSEIF *[~]expression*
ENDIF
ENDC

See Also **CASE**

Remarks

- The **ENDC** and **ENDIF** directive are interchangeable.
- If the **ELSEIF** directive is used without a following expression, it acts exactly the same as an **ELSE** directive.
- The optional tilde, preceding the operand expression, is synonymous with **NOT**. Its use normally necessitates the prudent use of brackets to preserve the sense of the expression.

Examples

```
                if      TargetA
sec_dir         equ     2
                elseif  TargetB
sec_dir         equ     1
                else
sec_dir         equ     3
                endif
```

```
                if      ~usesquare
round           macro
                lda     \1
                ...
                endm

                elseif

round           macro
                endm
                endc
```

CASE and ENDCASE

Description The **CASE** directive is used to select code in a multiple-choice situation. The **CASE** argument defines the expression to be evaluated; if the argument(s) after the *equals sign* are true, the code that follows is assembled. The *equals-question mark* case is selected if no previous case is true.

Syntax

CASE *expression*
=*expression*[,*expression*]
=?
ENDCASE

See Also **IF**

Remarks

- In the absence of a *equals-question mark (=?)* case, if the existing cases are unsuccessful, the case-defined code is not assembled.

Examples

```
Version     equ     Live
            ...
            case    Version
=Test,Demo
            lda     #$21
            sta     $4301
            jsr     SetPLevel
            rts
```

```
=Live
                lx      #Tstack
                ...
                jmp     SetLevels
=?
                inform  3,"Unknown Version"
                endcase
```

The following is an alternative for the example listed under the IF directive - see page 79:

```
Target          equ     TargetA
                ...
                case    Target
=TargetA
sec_dir         equ     2
=TargetB
sec_dir         equ     1
=?              db      "New Version",0
sec_dir         equ     3
                endcase
```

REPT, ENDR

Description These directives allow the programmer to repeat the code between the **REPT** and **ENDR** statements. The number of repetitions is determined by the value of *count*.

Syntax

```
REPT    count
...
ENDR
```

See Also **DO, WHILE**

Remarks

- When used in a Macro, REPT is frequently associated with the NARG function.

Examples

```
        rept    12
        dw      0,0,0,0
        endr
```

```
cbb     macro   string
lc      =       0
        rept    strlen(\string)
cc      substr  lc+1,lc+1,\string
        db      "\cc"^($A5+lc)
lc      =       lc+1
        endr
        endm
```

WHILE, ENDW

Description These directives allow the programmer to repeat the code between the **WHILE** and **ENDW** statements, as long as the expression in the operand holds true.

Syntax

```
WHILE   expression
...
ENDW
```

See Also **REPT, DO**

Remarks

- Currently, any string equate substitutions in the **WHILE** expression take place once only, when the **WHILE** loop is first encountered - see Note[1] below for the ramifications of this.

Examples

```
MultP   equ     16
        ...
Indic   =       MultP
        while   Indic>1
        lda     #$8f
        sta     $2100
        stz     $420c
        ...
Indic   =       Indic-1
        endw
```

Note[1]

- Because string equates are only evaluated at the start of the **WHILE** loop, the following will not work:

```
s       equs    "x"
        while   strlen("\s") < 4
        db      "\s",0
s       equs    "\s\x"
        endw
```

To avoid this, set a variable each time round the loop to indicate that looping should continue:

```
s       equs    "x"
looping =       -1
        while   looping
        db      "\s",0
s       equs    "\s\x"
looping =       strlen("\s") < 4
        endw
```

DO, UNTIL

Description These directives allow the programmer to repeat the code between the **DO** and **UNTIL** statements, until the specified expression becomes true.

Syntax

DO
...
UNTIL *expression*

See Also **REPT, WHILE**

Remarks Unlike the **WHILE** directive, string equates in an **UNTIL** expression will be re-evaluated each time round the loop.

Examples

```
MultP   equ     16
        ...
Indic   =       MultP
        do
        lda     #$8f
        sta     $2100
        ...
Indic   =       Indic-1
        until   Indic<=1
```

Register Handling Directives

These directives allow the ASM658 Assembler to optimise access to variables and extend the options available when handling 65816 registers:

- **ASSUME**
- **PROC**
- **ENDP**
- **LABEL**
- **CALL**
- **JUMP**
- **LONGA**
- **LONGI**
- **MX**
- **PUSHA**
- **POPA**

The following directive allows the programmer to specify certain initial parameters in the target machine:

- **REGS**

ASSUME

Description The **ASSUME** statement allows the assembler to optimise access to the programs variables.

Syntax **ASSUME** *register:value{,register:value}*

Remarks

- Valid registers are :

D	Direct Page Register
DB	Data Bank Register

- The value can be :

?	Register value not known
value	Register contains specified value
section name	Register points at the base of the specified section
group name	Register points at the base of the specified group

- More than one assume can be specified by separating the values by commas; for example:

```
assume    d:sect1,db:group1
```

where Direct Page points to base of sect1, Data Bank points to base of group1.

- If the Direct Page Register value is known, the Assembler can automatically generate direct page references:

```
assume   d:$300
lda      $323
```

will generate a reference to direct page location $23.

If variable 'var' is in section 'sect1', then:

```
assume   db:sect1
lda      var
```

will generate an absolute reference to 'var'

- It is always possible to force a particular type of addressing mode by using the < | and > prefixes on the operand.

- The Assembler has no way of checking that the values specified in the ASSUME statement are correct.

PROC, ENDP

Description The Assembler and Linker have the facility to check that various registers are set to the required values when a subroutine is called, or a label is jumped to. This is achieved by use of the **PROC** and **ENDP** directives.

Syntax

label **PROC** *[option{,option}]*
...
ENDP

See also **MX**

Remarks

- The label must be present, as the name of the subroutine.

- More than one option can be specified, if separated by commas. The following options are available :

near	Subroutine is to be called by JSR, not JSL
far	Subroutine is to be called by JSL, not JSR
mx:value	Values of the M and X bits of the status register
d:value	Value in the Direct Page Register
db:value	Value in the Data Bank Register

  - The values for the **mx** option can be 0, 1, 2 or 3, as in the **MX** statement (see page 96).

  - The values for the Direct Page Register are the same as can be specified in an **ASSUME** statement (see page 89).

- The values for the Data Bank Register are the same as can be specified in an **ASSUME** statement.

- At the PROC statement, the Assembler saves the current MX values, together with the currently assumed values for the Data Bank and Direct Page Registers. It then sets these values to those specified in the PROC statement. Any options that are unspecified will be left unchanged.

- At the ENDP statement, the MX value, the Data Bank and Direct Page ASSUME values will be restored.

- PROCs may be nested. The Assembler will check that the appropriate type of return instruction is used inside a PROC. If a PROC is specified as **near**, the Assembler will give a warning if an RTL instruction is used.

LABEL

Description A **LABEL** statement is similar to a PROC statement except that it does not save the current states of the MX value, the Data Bank or Direct Page Registers. There is no corresponding *endlabel* statement.

Syntax *label* **LABEL** *[option{,[option]}]*

See also **PROC**

Remarks

- The label statement will normally be used for alternative entry points into subroutines; frequently, therefore, there are no options specified. If options are included, they take the same form as for the PROC directive.
- The LABEL statement notes the current settings of the MX value, together with the Data Bank and Direct Page Registers, and whether it is in a **near** or **far** callable subroutine. When the LABEL statement is called, these values will be checked against those that are currently set.

CALL, JUMP

Description In order that the Assembler can check that registers contain the appropriate values, a **CALL** statement must be used to access the subroutine.

Syntax

CALL *subroutine*

JUMP *subroutine*

Remarks

- The assembler will check that the MX value, and the values of the Data Bank and Direct Page Registers, are as specified in the PROC statement. If any do not correspond, an error will be generated.

- If an option is not specified in the PROC statement, no check will be made for that value.

- If either the Data Bank or Direct Page Register was specified as ? (unknown value),no check will be made on that register. Note that, other than as a result of ***rep*** and ***sep*** instructions, the Assembler cannot check that values specified in ASSUME and MX statements are actually correct.

- If a subroutine is called which is already defined, the Assembler will generate a ***jsr*** or ***jsl*** instruction, as appropriate, as long as a **near** or **far** option was specified in the PROC statement. If the subroutine is not yet

defined, or if no **near** or **far** option was specified, the Assembler will normally generate a ***jsr*** instruction. This can be overridden by specifying Long or Word Absolute Addressing in the CALL statement:

```
call    >sub1

call    |sub1
```

- If the subroutine has the **near** or **far** options specified, it will still check that the appropriate addressing mode has been used, and give an error if not; for instance:

```
Sub1    proc    near
        ...
        endp
        ...

        call    >Sub1
```

will give an error on the CALL statement.

- The **JUMP** statement is similar to the CALL statement; the same checks are performed but either a ***jmp*** or ***jml*** instruction will be generated.

LONGA, LONGI, MX

Description These statements allow the Assembler to be informed of the current size of the Accumulator and Index Registers, so that it can generate immediate operands of the correct size.

Syntax

MX *value*

LONGA *on/off*
LONGI *on/off*

Remarks

- The value of the MX statement specifies the settings of the M and X bits in the Processor Status Register. Possible values are :

0	Accumulator and Index Registers are all 16 bits
1	Accumulator is 16 bits, Index Registers are both 8 bits
2	Accumulator is 8 bits, Index Registers are both 16 bits
3	Accumulator and Index Registers are all 8 bits

- LONGA and LONGI take the following values:

LONGA ON	Accumulator is 16 bits
LONGA OFF	Accumulator is 8 bits
LONGI ON	Index Registers are 16 bits
LONGI OFF	Index Registers are 8 bits

- The Assembler will also note changes to the sizes of the registers when ***rep*** and ***sep*** instructions are used, although this is only done if the operand value can be evaluated on the first pass.

- The current MX value can be accessed by referring to the **__mx** variable. e.g.

```
if          __mx&1   ; if index registers
                     ;are 8 bit
...
endif
```

PUSHA, POPA

Description These directives allow the current ASSUME settings, for the Direct Page and Data Bank Registers, to be saved and restored.

Syntax

PUSHA

POPA

Example

```
pusha              ; save current settings
assume    db:$7e
...
popa               ; restore old  values
```

REGS

Description If a CPE file is produced, or object code sent directly to the target machine, the **REGS** directive specifies the values of the registers, at the start of code execution.

Syntax **REGS** *regcode=expression*[,*regcode=expression*]

where *regcode* is the mnemonic name of a register, as follows:

A	The Accumulator
X	Index Register X
Y	Index Register Y
S	The Stack Pointer
P	Processor Status Register
PC	The Program Counter
D	Direct Page Register
DB	Data Bank Register

Remarks

- This feature *cannot* be used if machine specific relocatable code is produced, or code in pure binary format.

Examples

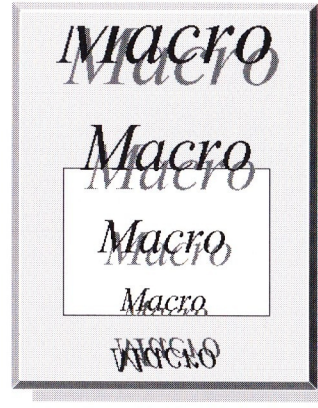
```
regs    s=$2700

regs    pc=entrypoint
```

A typical use for this directive is to set the Program Counter to the address for the start of execution, and the corresponding value of the stack.

CHAPTER 5

Macros



The **ASM658** assembler provides the programmer with extensive macro facilities. Macros allow the programmer to assign names to complete code sequences. They may then be used in the main program like existing assembler directives.

This chapter discusses the following topics, directives and functions:

- MACRO, ENDM
- MEXIT
- Macro Parameters
- SHIFT, NARG
- MACROS
- PUSHP, POPP
- PURGE
- TYPE

MACRO, ENDM, MEXIT

Description

A *macro* consists of the source lines and parameter place markers between the **MACRO** directive and the **ENDM**. The *label* field is the symbolic name by which the macro is invoked; the operand allows the entry of a string of parameter data names.

When the assembler encounters a directive consisting of the *label* and optional parameters, the source lines are pulled into the main program and expanded by substituting the place markers with the invocation parameters. The expansion of the macro is stopped immediately if the assembler encounters a **MEXIT** directive.

Syntax

Label **MACRO** [*symbol,..symbol*]
...
MEXIT
...
ENDM

See Also

MACROS

Remarks

- Control structures within macros must be complete. Structures, started in the macro, must finish before the **ENDM**; similarly, a structure started externally must not be terminated within the macro. To imitate a simple control structure from another assembler, a short macro might be used - see **MACROS**, page 111.

Examples

```
remove      macro
            dw          -2,0,0
            endm
```

```
Form        macro

            if          strcmp('\1','0')
            dw          0
            else
            dw          \1-FormBase
            endif

            endm
```

Macro Parameters

Parameters Macro parameters obey the following rules:

- The parameters listed on the macro invocation line may appear at any point in the code declared between the MACRO and ENDM statements. Each parameter is introduced by a backslash (\); where this may be confused with text from an EQUS, a backslash may also follow the parameter.

- Up to thirty two different parameters are allowed, numbered **\0** to **\31**.

- Instead of the **\0** to **\31** format, parameters can be given symbolic names, by their inclusion as operands to the **MACRO** directive. The preceding backslash (\) is not mandatory; however, if there is the possibility of confusion with the surrounding text, a backslash may be used before the symbol name, and, if necessary, after it, to ensure the expression is expanded correctly:

Example

```
Position    macro   A,B,C,Pos,Time
            dsw     \Time*(\A*\Pos,\B*\Pos,\C*\Pos)
            endm
```

- Surrounding the operand of an invoked macro with *greater than* and *less than* signs (**<...>**), allows the use of comma and space characters. This does not apply to assemblers which use angle brackets as address mode specifiers, such as ASM658; in these instances, backward single quote is used:

Example

```
Credits     macro
            dw   \1,\2
            db   \3
            db   0
            endm
            ...
            Credits     11,10,'Psy-Q, by Psygnosis'
```

- Continuation Lines - when invoking a macro, it is possible that the parameter list will become overlong. As with any directive statement, the line can be terminated by an ampersand **(&)** and continued on the next line to improve readability.

Example

```
chstr       macro
            rept        narg
            db          k_\1
            shift
            endr
            db   0
            endm
            ...
cheatstring chstr       i,c,a,n,b,a,r,e,l,y,&
                        s,t,a,n,d,i,t
```

Special Parameters

There are a number of special parameter formats available in macros, as follows:

Converting Integers to Text

The parameters \# and \$ replace the decimal (#) or hex ($) value of the symbol following them, with its character representation. Commonly, this technique is used to access Run Date and Time:

Example

```
RunTime     db   "\#_hours:\#_minutes:&
                    \#_seconds"
```

this expands to the form *hh:mm:ss*, as follows

```
RunTime     db          "21:08:49"
```

Generating Unique Labels

The parameter \@ can be used as the last characters of a label name in a macro. When the macro is invoked, this will be expanded to an underscore followed by a decimal number; this number is increased on each subsequent invocation to give a unique label.

Example

```
Init        macro
ent\@       ...
            ldx         \1,y
            bpl         ent\@
```

```
                ldy     #0
                bra     ent2\@
set2\@          lda     \2+1,y
                sta     >$004200,x
                iny
                iny
ent2\@          ldx     \1,y
                bpl     set2\@
                endm
                ...
                Init    initregs21,initregs42
```

Each time the *Init* macro is used, new labels in the form ent_001, ent2_001 and set2_001 will be generated.

Entire Parameter

If the special parameter _ (backslash underscore) is encountered in a macro, it is expanded to the complete argument specified on the macro invocation statement.

Examples

```
All             macro
                db      \_
                endm
                ...
                All     1,2,3,4
```

will generate

```
                db      1,2,3,4
```

Control Characters

The parameter \^**x**, where **x** denotes a control character, will generate the specified control character.

Using the Macro Label

The label heading the invocation line can be used in the macro, by specifying the first name in the symbol list of the **MACRO** directive to be an asterisk (*), and substituting * for the label itself. However, the resultant label is not defined at the current program location. Therefore, the label remains *undefined* unless the programmer gives it a value.

Extended Parameters

The **ASM658** Assembler accepts a set of elements, enclosed in curly brackets ({ }), to be passed to a macro parameter. The NARG function and SHIFT directive can then be used to handle the list - see also page 110:

Example

```
cmd     macro
cc      equs    {\1}
        rept    narg(cc)
         \cc
        shift   cc
        endr
        endm
```

SHIFT, NARG

Description These directives cater for a macro having a variable parameter list as its operand. The **NARG** symbol is the number of arguments on the macro invocation line; the **SHIFT** directive shifts all the arguments one place to the left, losing the leftmost argument.

Syntax

directive **NARG**
...
SHIFT

where **NARG** is a reserved, predefined *symbol*.

See Also **Extended Parameters**

Examples

```
routes      macro
            rept        narg
            if          strcmp('\1','0')
            dw          0
            else
            dw          \1-routebase
            endif
            shift
            endr
            endm
            ...
            routes      0,gosouth_1
```

MACROS

Description The **MACROS** directive allows the entry of a single line of code, as a macro, with no associated **ENDM** directive. The single line of code can be a control structure directive.

Syntax *Label* **MACROS** [*symbol,..symbol*]

See Also **MACRO**

Remarks

- The **MACROS** directive may be used to stand in for a single, complex code line. Often, the short macro allows the programmer to synthesise a directive from another assembler. Including the /k option on the ASM658 command line, will cause several macros which emulate foreign directives to be generated.

Examples

```
                if      0
boom            macros
                beq     noboom
                else
boom            macros
                lda     #blowup-tactbase,&
                        slot_tactic(\1)
                endif
```

PUSHP, POPP

Description These directives allow text to be pushed into, and then popped from, a string variable.

Syntax

PUSHP *string*
POPP *string*

Remarks There is no requirement for the **PUSH** and corresponding **POPP** directive to appear in the same macro.

Examples

```
ifhid   macro
        dw      ibvis
        dw      \1
        pushp   "\@"
        dw      \@-2-*
        endm
        ...
ifnot   macro
        popp    lab
        goto    \@
        pushp   "\@"
\lab
        endm
```

PURGE

Description The **PURGE** directive removes an expanded macro from the symbol table and releases the memory it occupied.

Syntax **PURGE** *macroname*

Remarks If it is required to redefine a macro, it is not necessary to purge it first. If an existing macro is redefined, the original definition is purged by the Assembler first.

Examples

```
HugeM   macro
        dw      \1
        ...
        dw      \31
        endm

        HugeM   para1,103,faultlevel,&
        ...
                40,50,para31

        purge   HugeM
```

TYPE

Description TYPE is a function to provide information about a symbol. It is frequently used with a Macro to determine the nature of its parameters. The value is returned as a word; the meanings of the bit settings are given below.

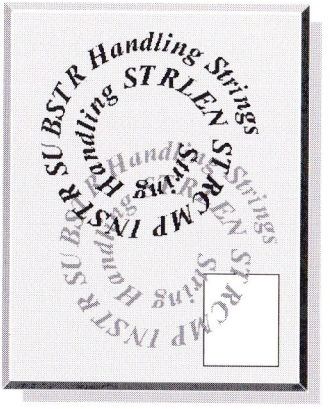
Syntax **TYPE**(*symbol*)

The reply word can be interpreted as follows:

Bit 0	Symbol has an absolute value
Bit 1	Symbol is relative to the start of the Section
Bit 2	Symbol was defined using **SET**
Bit 3	Symbol is a Macro
Bit 4	Symbol is a String Equate (**EQUS**)
Bit 5	Symbol was defined using **EQU**
Bit 6	Symbol appeared in an **XREF** statement
Bit 7	Symbol appeared in an **XDEF** statement
Bit 8	Symbol is a Function
Bit 9	Symbol is a Group Name
Bit 10	Symbol is a Macro parameter
Bit 11	Symbol is a short Macro (**MACROS**)
Bit 12	Symbol is a Section Name
Bit 13	Symbol is Absolute Word Addressable

CHAPTER 6

String Manipulation Functions



To enhance the Macro structure, the **ASM658** assembler provides powerful functions for string manipulation. These enable the programmer to compare strings, examine strings and prepare subsets.

This chapter covers the following string handling functions and directive:

- STRLEN
- STRCMP
- INSTR
- SUBSTR

STRLEN

Description A function which returns the length of the text specified in the brackets.

Syntax **STRLEN**(*string*)

See Also **STRCMP**

Remarks

- The **STRLEN** function is available at any point in the operand.

Examples

```
Nummov  macro

        rept    strlen(\1)
        lda     #2
        sta     $420b
        endr
        endm
        ...
        Nummov  "12345"
```

The number of characters in the string is used as the extent of the loop.

STRCMP

Description A function which compares the two text strings in the brackets, and returns *true* if they match, otherwise it returns *false*.

Syntax **STRCMP**(*string1*,*string2*)

See Also **STRLEN**

Remarks

- When comparing two text strings, the **STRCMP** function starts numbering the characters in the target texts from one.

Examples

```
Vers    equ     "Acs"
        ...
        if      strcmp("\Vers","Sales")
        lda     SalInd
        else
        if      strcmp("\Vers","Acs")
        lda     AcInd
        else
        if      strcmp("\Vers","Test")
        lda     TstInd
        endif
        endif
        endif
```

INSTR

Description A function which searches a text string for a specified sub-string. If the string does not contain the sub-string, a result of zero is returned, if the sub-string is present, the result is the location of the sub-string from the start of the target text. There is an optional parameter specifying an alternate start point within the string.

Syntax **INSTR** ([*start,*]*string*, *sub-string*)

See Also **SUBSTR**

Examples

```
Mess    equs    "Demo for Sales Dept"
        ...
        if      instr("\Mess","Sales")
        lda     SalInd
        else
        lda     AcInd
        endif
```

Note

- When returning the offset of a located sub-string, the **INSTR** function starts numbering the characters in the target text from one.

SUBSTR

Description A directive assigns a value to a symbol, which is a sub-string of a previously specified text string, defined by the *start* and *end* parameters. The *start* and *end* parameters will default to the start and end of the string, if omitted.

Syntax *symbol* **SUBSTR** [*start*],[*end*],*string*

See Also **INSTR, EQUS**

Examples

```
Message     equs    "A short Sample String"
            ...
Part1       substr  9,14,"\Message"
Part2       substr  16,,"\Message"
Part3       substr  ,7,"\Message"
Part4       substr  ,,"\Message"
```

where Part1 equals *Sample*
Part2 equals *String*
Part3 equals *A short*

The last statement is equivalent to an EQUS assigning the whole of the original string to Part4.

```
cbb         macro   string
lc          =       0
            rept    strlen(\string)
cc          substr  lc+1,lc+1,\string
            db      '\cc'^($A5+lc)
lc          =       lc+1
            endr
            endm
```

CHAPTER 7

Local Labels

As a program develops, finding label names that are both unique and definitive becomes increasingly difficult. Local Labels ease this situation by allowing meaningful label names to be re-used.

This chapter covers the following topics and directives:

- Local Label Syntax and Scope
- MODULE and MODEND
- LOCAL

Syntax and Scope

Syntax

- Local Labels are preceded by a local label signifier. By default, this is an **@** sign; however, any other character may declared by using the l option in an **OPT** directive or on the Assembler command line - see **Assembler Options**, page 141.
- Local label names follow the general label rules, as specified on page 24.
- Local labels are not de-scoped by the expansion of the macro.

Scope

The region of code within which a Local Label is effective is called its *Scope*. Outside this area, the label name can be re-used. There are three methods of defining the scope of a Local Label:

- The scope of a local label is implicitly defined between two non-local labels. Setting a variable, defining an equate or RS value does not de-scope current local labels, unless the **d** option has been used in an **OPT** directive or on the Assembler command line - see **Assembler Options**, page 141.
- The scope of a Local Label can also, and more normally, be defined by the directives **MODULE** and **MODEND** - see page 125.
- To define labels (or any other symbol type) for local use in a macro, the LOCAL directive can be used - see page 121.

Examples

```
plot2       lda     compw
            ...
            ldx     HiSpot,y
            bpl     @chk1
            ldy     #0
            bra     @ret
@chk1       lda     HiSpot+1,y
            sta     >$004200,x
            iny
            iny
            ...
SetX        set     x+1
@ret        rts
plot3       lda     HiSpot+2,y
            ...
@ret        rts
```

The code above shows a typical use for Local Labels, as "place markers" within a self-contained sub-routine. The scope is defined by the non-local labels, *Plot2* and *Plot3*; the **SET** statement does not de-scope the routine. The labels *@chk1* and *@ret* are re-usable.

```
Lab1        lda     |$0,x
            bne     @add
            ...
@add        ldx     HiSpot,y
            ...
Lab2        ldy     #0
            bra     @add
```

In this example, the final branch will cause an error, since it is outside the scope of @add.

MODULE and MODEND

Description Code occurring after a **MODULE** statement, and up to *and including* the **MODEND** statement , is considered to be a *module*. Local labels defined in a *module* can be re-used, but cannot be referenced outside the module's scope. A Local label defined elsewhere cannot be referenced within the current module.

Syntax

```
MODULE
...
...
MODEND
```

See Also LOCAL

Remarks

- Modules can be nested.
- The **MODULE** statement itself is effectively a non-local label and will de-scope any currently active default scoping.
- Macros can contain modules or be contained in a module. A local label occurring in a module can be referred to by a macro residing anywhere in that module. A module contained within a macro can effectively provide labels local to the macro.

Examples

```
KillCar     module

            jsr   @FindCar
            inx
            ...
@FindCar    lda       HiCar+1,y
            ...

TryCar      module
            lda       |$0,x
            bne       KillCar
            rts

            modend  TryCar

            modend  KillCar
```

LOCAL

Description The **LOCAL** directive is used to declare a set of macro-specific labels and other symbols.

Syntax **LOCAL** *symbol,..,symbol*

See Also **MODULE**

Remarks

- The scope of symbols declared using the **LOCAL** directive is restricted to the host macro.
- The **LOCAL** directive does not force a type on the symbols that make up its parameters. In practice, therefore, such symbols can be declared as equates, string equates or any other type, as well as labels.

Examples

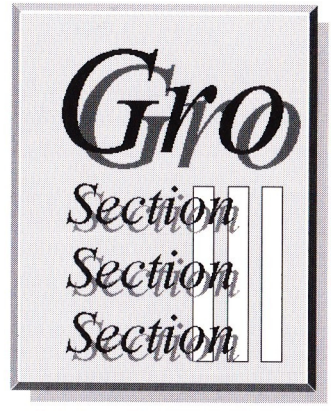
```
MacShow     macro

            local   Lab1,Numb1,Numb2,Text1
            bra     Lab1

Numb1       equ     63
Numb2       equ     15
Text1       equs    "\A\\B"
Lab1        ...
            endm
```

CHAPTER 8

Structuring the Program



Normally, the organisation of the memory of the target machine does not match the layout of the source files. To create a structured target memory, as well as to create relocatable program sections, the **ASM658** assembler uses the concept of Groups and Sections.

This chapter covers the following topics and directives:

- SECTION
- GROUP
- PUSHS and POPS
- SECT and OFFSET

GROUP

Description This directive declares a group with up to six group attributes.

Syntax *GroupName* **GROUP** [*Attribute,..Attribute]*

where an attribute is one of the following - see below for descriptions:

BSS
ORG(*address*)
FILE(*filename*)
OBJ(*address*)
SIZE(*size*)
OVER(*GroupName*)

See Also **SECTION**

Remarks

- *Group Attributes* are interpreted as follows:

BSS - no initialised data. to be declared in this group.

Example

```
Group1      group      bss
```

ORG - sets the ORG address without reference to the other group addresses. If this attribute is omitted, the group will be placed in memory, following on from the end of the previous group.

Example

```
                org         $100
G1              group
G2              group       org($400)
G3              group
```

will place the groups in the sequence G1,G2,G3

FILE - outputs a group, such as an overlay, to its own binary file; other groups will be output to the declared file.

Example

```
Group1          group       org($400),file("ov1.bin")
```

OBJ - sets the group's OBJ address. Code is assembled as if it is running at the OBJ address but is placed at the group's ORG address. If no address is specified then the OBJ value is the same as the group's ORG address.

Examples

```
Group1          group       org($400),obj($1000)
Group2          group       org($800),obj()
```

SIZE - specifies the maximum allowable size of the group. If the size exceeds the specified size, the Assembler reports an error.

Example

```
Group1          group       size(32768)
```

The SIZE attribute has a second optional parameter, to specify the value that should be used to pad the group if it is less than the specified size. In the example below, Group is allowed a size of 32k and unused bytes are padded with $FF.

Example

```
Group1      group     size(32768,$ff)
```

OVER - overlays this group on the specified group. Code at the start of the second group is assembled at the same address as the start of the first group. The largest of the overlayed groups' sizes is used as the size of each group. Note that it is necessary to use the FILE attribute to force different overlays to be written to different output files.

Example

```
Group2      group     over(Group1)
```

SECTION

Description This directive declares a logical code section.

Syntax

SECTION *SectionName*[,*Group*]

SECTIONB *SectionName*[,*Group*]
SECTIONW *SectionName*[,*Group*]

SectionName **SECTION** *[Attribute,..Attribute]*

The last format is a special case, designed to allow definition of a section with group attributes - see below for a description.

See Also **GROUP**

Remarks

- By default, SECTION declares a byte-aligned section of the source code. SECTIONB and SECTIONW allow a section to be forcibly byte- or word-aligned.
- Unless the section has been *previously* assigned, the section will be placed in an unnamed default group, if the GROUP name is omitted
- It is possible to define a section with group attributes. The assembler will automatically create a group with the section name preceded by a tilde (~) and place the section in it.

Example

```
Sect1           section     bss
```

defines Sect1, with the BSS attribute, in a group called *~Sect1*.

- Note that, if a section is *byte* aligned that the **CNOP** directive cannot be used to align the PC to a value which is larger than the alignment of the current section - see pages 66.

- Iff sections are used to structure application code, only a single ORG directive ican be used, which must precede all section definitions. Groups and Sections may have ORG *attributes* to position them.

  When producing linkable output, no ORG directives or attributes are permitted. Sections are ordered, within a group, in the sequence that the Linker encounters the section definitions.

- It can also be left to the assembler to assign a section to any group which has room for it, by specifying **?** as the group name.

  ```
  section     s1,?
  ```

  The section s1 will be placed in any group with room, although only groups that have had a size specified will be considered as candidates. This is useful facility on the SNES, since memory is broken up into 32K or 64K banks and the programmer may not care which bank a section is actually in.

Examples

This example shows the use of Groups and Sections to impose a structure on the target memory, as follows:

- preliminary version checks and includes;
- group declarations;
- a section of application code;
- a section of uninitialised data.

```
                opt         c-,s,v+

version         equ         0       ; 0 => full version
                                    ; 1 => demo version
                                    ; 2 =>  test version

                include     "miscmac.obj"
                include     "rooms.obj"
                include     "output.obj"

                org         $100
numvecs         equ         $100>>2
                regs        pc=progstart

                 if         ~def(amiga)
amiga           equ         1
                endif

                 if         ~def(ntsc)
ntsc            equ         1
                endif

g_code          group
g_bss           group       bss

                section     code,g_code
                section     bss,g_bss

firstbss        equ          *
```

PUSHS and POPS

Description These directives allow the programmer to open a new, temporary section, then return to the original section. **PUSHS** saves the current section, **POPS** restores it.

Syntax

PUSHS

POPS

Examples

```
passdl      equ     *
            ...
            pushs
            section dolight
            dl      passdl
            ...
            pops
```

This example shows PUSHS and POPS being used to pass system information, in the form of the location counter, between sections.

SECT and OFFSET

Description The **SECT** function returns the address of the section in which the symbol in the brackets is defined. The **OFFSET** function returns the location, in the host section, of the symbol in the brackets.

Syntax

SECT (*expression*)

OFFSET (*expression*)

Remarks

- If a link is being performed, the **SECT** function is evaluated when it is linked; if there is no link, it will be evaluated when the second pass has finished.
- Likewise, if a link is being performed, the **OFFSET** function is evaluated when it is linked; however, if there is no link, it will be evaluated during the first pass.

Examples

```
dw      sect(Table1)
dw      offset(Table2)
dw      offset(*)
```

CHAPTER 9

Options, Listings and Errors

This chapter completes the discussion of the **ASM658** Assembler and its facilities. It covers methods of determining run-time Assembler options, producing listings and error-handling, as well as passing information to the Linker:

- OPT
- Assembler Options
- PUSHO and POPO
- LIST and NOLIST
- INFORM
- FAIL
- XREF, XDEF and PUBLIC
- GLOBAL

OPT

Description This directive allows Assembler options to be enabled or disabled in the application code - see overleaf for a full list of options.

Syntax **OPT** *option,..option*

See Also **PUSHO, POPO**

Remarks

- An option is turned on and off by the character following the option code:

  \+ (plus sign) = *ON*

  \- (minus sign) = *OFF*

- Options may also be enabled or disabled by using the **/O** switch on the Assembler command line - see Command Line Syntax, page 13.

Examples

```
opt     an+,l:+,e-

opt     w+,ws+,od+
```

Assembler Options

The following reference list shows the various options and optimisations available during assembly, with the default settings; the options are later described in detail.

Option	*Default*	*Description*
20	Produce binary output for Mode 20 ROM	Off
AN	Enable Alternate Numeric mode	Off
B	Swap meaning of < and >	Off
C	Activate/ Suppress Case sensitivity	Off
D	Allow EQU or SET to descope local labels	Off
E	Disable error text print	On
L*x*	Substitute *x* local label signifier	Off
S	Handle equated names as labels	Off
T	Truncate operand values automatically	Off
TA	Truncate address operands	Off
V	Write Local Labels to symbol file	Off
W	Disable warning messages	On
WS	Operands may contain white space	Off
X	XREFs in defined section	Off
Optimisation:		
OD	Direct Page Optimisation	Off

- Note that Assembler Optimisation can only be performed on backward references.

Option Descriptions

20 - Output for Mode 20

Using the 20 option causes the Assembler to produce pure binary output for the Mode 20 ROM.

AN - Alternate Numeric

Setting this option allows the inclusion of numeric constants in Zilog or Intel format; that is, followed by H, D or B, to signify Hex, Decimal or Binary. See also the section on the RADIX directive - page 37.

B - Swap < and >

This option swaps the the meanings of *less than* and *greater than*, when used in expressions.

C - Case Sensitivity

When the C option is set, the case of the letters in a label's name is significant. For example, *SHOWSTATS*, *ShowStats* and *showstats* could all be legally used.

D - Descope Local Labels

If this option is enabled, local labels will be descoped if an EQU or SET directive is encountered.

E - Error Line Printing

If this option is enabled, the text of the line that caused an Assembler error will be printed, as well as the host file

name and line number. *If this option is not specified, the default is ON.*

L- Local Label Character

In **ASM658**, local labels are signified by a preceding AT sign (@). This option allows the use of the character following the option letter as the signifier. Thus, **L:** would change the local label character to a colon (:). **L+** and **L-** are special formats that toggle the character between an exclamation mark (!) and an @ sign (-).

T - Truncate operand values

When this option is used, operand values are automatically truncated by the Assembler.

TA - Truncate address operands

When this option is used, address operands are automatically truncated by the Assembler.

V - Local Labels to Symbol file

The Assembler will output all Local Label names to the nominated symbol file, if this option is used.

W - Give warnings

The Assembler identifies various instances where a warning message would be printed, but assembly allowed to continue. Disabling the W option will suppress the reporting of warning messages. *If this option is not specified, the default is ON.*

WS - Allow white spaces

If this option is set ON, operands may contain white spaces. Thus, the statement:

```
dl          1 + 2
```

defines a value of 1 with WS set OFF, and a value of 3 with WS set to ON.

X - XREFs in defined section

This option set to ON specifies that XREFs are assumed to be in the section in which they are defined.

OD - Direct Page Optimisation

This option forces the Assembler to utilise Direct Page optimisation.

PUSHO and POPO

Description The **PUSHO** directive saves the current state of all the assembler options; **POPO** restores the options to their previous state. They are used to make a temporary alteration to the state of one or more options.

Syntax

PUSHO

POPO

See Also **OPT**

Examples

```
                pusho
                opt     ws+, c+

SetAlts         =       height*time
SETALTS         dw      256 * SetAlts

                popo
```

LIST and NOLIST

Description The NOLIST directive turns off listing generation; the LIST directive turns on the listing.

Syntax

NOLIST

LIST *indicator*

where indicator is a plus sign (+) or a minus sign (-).

Remarks

- If a list file is nominated, either by its inclusion on the command line, or in the ASM658 environment variable, the a listing will be produced during the first pass.
- The Assembler maintains a *current listing status* variable, which is originally set to zero. List output is only generated when this variable is zero or positive. The listing directives affect the listing variable as follows:

- **NOLIST** sets it to -1;
- **LIST** with no parameter, zeroises it;
- **LIST** + adds 1;
- **LIST** - subtracts 1.

Examples

Directive	*Status*	*Listing produced?*
nolist	-1	no
list -	-2	no
list	0	yes
list -	-1	no
list -	-2	no
list +	-1	no
list +	0	yes

Note

In the following circumstances, the Assembler automatically suppresses production of listings:

- during macro expansion;
- for unassembled code because of a failed conditional.

These actions can be overridden by:

- including the **/M** option on the Assembler command line to list expanding macros;
- including the **/C** option on the Assembler command line to list conditionally ignored code - see Command Line Syntax, page 13.

INFORM and FAIL

Description The INFORM directive displays an error message contained in *text*, which may optionally contain parameters to be substituted by the contents of *expressions*, after evaluation. Further Assembler action is based upon the state of *severity*.

Syntax

INFORM *severity,text*[,*expressions*]

FAIL *(see Note[1])*

Remarks

- These directives allow the programmer to display an appropriate message if an error condition, which the Assembler does not recognise, is encountered
- ***Severity*** is in the range 0 to 3, with the following effects:

- **0** : the Assembler simply displays the text;
- **1** : the Assembler displays the text and issues a warning;
- **2** : the Assembler displays the text and raises an error;
- **3** : the Assembler displays the text, raises a fatal error and halts the assembly.

- ***Text*** may contain the parameters %d, %h and %s. They will be substituted by the decimal, hex or string values of the operands.

Examples

```
TableSize   equ     TableEnd-TableStart
MaxTable    equ     512
            if      TableSize>MaxTable
            inform  0,"Table starts at %h and&
                     is %h bytes long",&
                    TableStart,Tablelen
            inform  3,"Table Limit Violation"
            endif
```

Note[1]

- The FAIL directive is a pre-defined statement, included for compatibility with other assemblers. It generates an "Assembly Failed" message and halts assembly.

XDEF, XREF and PUBLIC

Description If several sub-programs are being linked, to refer to symbols in a sub-program which are defined in another sub-program, use **XDEF**, **XREF** and **PUBLIC**.

Syntax

XDEF *symbol*[,*symbol*]
XREF *symbol*[,*symbol*]

PUBLIC *on*
PUBLIC *off*

Remarks

- In the sub-program where symbols are initially defined, the **XDEF** directive is used to declare them as externals.
- In the sub-program which refers the symbols, the **XREF** directive is used to indicate that the symbols are in a another sub-program.
- The Assembler does not completely evaluate an expression containing an **XREF**ed symbol; however, resolution will be effected by the linker.
- The **PUBLIC** directive allows the programmer to declare a number of symbols as externals. With a parameter of ***on***, it tells the Assembler that all further symbols should be automatically **XDEF**ed, until a **PUBLIC** ***off*** is encountered.

Examples Sub-program A contains the following declarations:

```
        xdef    Scores,Scorers
        xref    PointsTable
        ...
```

The corresponding declarations in sub-program B are:

```
        xdef    PointsTable
        xref    Scores,Scorers
        ...
```

```
        public  on
Origin  =       Mainchar
Force   dw      speed*origin
Rebound dw      45*angle
        public  off
```

GLOBAL

Description The **GLOBAL** directive allows a symbol to be defined which will be treated as either an **XDEF** or an **XREF**. If a symbol is defined as **GLOBAL** and is later defined as a label, it will be treated as an **XDEF**. If the symbol is never defined, it will be treated as an **XREF**.

Syntax **GLOBAL** *symbol*[,*symbol*]

See Also **XREF**, **XDEF**, **PUBLIC**

Remarks This is useful in header files because it allows all separately assembled modules to share one header file, defining all global symbols. Any of these symbols later defined in a module will be **XDEF**ed, the others will be treated as **XREF**s.

CHAPTER 10

DBUG658 - The Debugger

DBUG658 is a full source level Debugger, as well as a traditional symbolic Debugger. This allows *source code* to be viewed, run and traced, stepped-over, breakpoints set and cleared.

The original symbolic debug facilities are all still available. A source level display will revert to a symbolic disassembly, when no source level information is available.

The following Debugger topics are discussed in this chapter:

- Command Line Syntax
- Configuration
- Activity Windows
- Debugger Options
- Menu and Keyboard Usage
- Link Software

Command Line Syntax

Syntax

DBUG658 */switches filename filename*

DBUG658 *?*

displays a help message.

Remarks

- *Filename* specifies the name of a file containing symbols, produced by the using the /zd option during assembly. If no extension is shown, a default extension of .SYM will be added. Multiple filenames are allowed and must be separated by a space - the symbol files will then be loaded in the order specified.

- Valid switches are:-

/h	Halt target at Debugger start-up.
/s*file*	Override default configuration filename - see page 160.
/v*exprtext*	Evaluate *exprtext* and put result to standard output device.
/e*file[,file,file]*	Load target machine with CPE file(s).
/r##	Specify data screen rows in video bios.
/c-	Turn case sensitivity off
/c+	Turn case sensitivity on
/i###	Specify update interval (in 1/18ths sec)
/t#	Set target SCSI device number (default is 0).
/u-	Turn continual update mode OFF (+ for ON)
/&*expr,.. expr*	List of parameter expressions, separated by commas.

/m#	Sets the Debugger mouse sensitivity; # is a number between 1 to 4 - default is 3. DBUG658 drives the mouse itself. This overcomes some shortcomings exhibited by the Microsoft Mouse driver, particularly in 132 column mode.
/m+	Use the current system mouse driver; later versions of the Microsoft drivers (8 upwards) allow the mouse to be used in a DOS window.
/m-	Revert to DBUG658 mouse driver.

Remarks

- Source level mode can be used if a symbol file is specified on the command line. This file contains symbols and additional source level information produced by the **/zd** option in the ASM658 Assembler - see page 13. See page 163 for more about Source Level Debugging.

- Expressions passed to the Debugger using the **/&** switch can be referred to in the form &0, &1, etc., where 0 means the first expression on the command line, 1 means the second.

- Label level values are as follows:

Label Level	*Meaning*
0	Display symbols only in address field.
1	Display symbols in address field, and show non-immediate or offset symbols.

2	Display as Level 1, and show long-immediate symbols.
3	Display as Level 2, and show offset (*An*) symbols.
4	Display as Level 3, and show offset (*An,Dn*) symbols.

Configuration Files

When **DBUG658** is loaded, it accesses a configuration file, containing information about the current Debugger environment. The current configuration can be saved at any time during an active Debugger session. The default filename can be overriden with an option on the command line (/s) or at run-time, so that the most frequently used configurations are always readily available.

Configuration File Names

- The normal configuration file name is DBUG658.C##, where the first number is the target SCSI id number, and the second number is the virtual screen. Typically, therefore, the configuration loaded at start-up is DBUG658.C00.

- If this file is not located in the current directory, the Debugger looks for a file called DBUG658.DF# (the default configuration file).

- If the default configuration cannot be found, the Debugger will search the directory from which the program was loaded, for a file called SNES____.CFG.

Contents of Configuration File

A configuration file can include the following information:

- Read Memory Ranges;
- Write Memory Ranges;
- Video type and usage;
- Label Level;
- Colour and Mono attributes;
- Tab Settings;
- Current Window Type and Display position;
- Breakpoints;
- History details.

Activity Windows

The Debugger display consists of one or more activity windows. The number of windows, the contents of each window and the window size, can all be specified at run-time. The default display consists of two windows; the upper window normally contains a display of the registers, the lower window shows the disassembly of the code at the current pointer.

The Debugger can run up to 10 virtual screens; each screen has its own configuration file - see page 160. Alternate screens can be accessed by pressing Alt-*n*, where *n* is the screen number 0 - 9, where 0 means screen 10.

Window Types

The **Register Window** provides a complete view of the selected processor's registers. Register contents can be changed:

- by typing directly at the current cursor location;
- by entering an expression; pressing the ENTER key displays an expression input window.

The **Disassembly Window** shows the contents of the target memory as disassembled code. If a symbol file has been loaded into the Debugger, symbol names are substituted as appropriate, according to the label level. Breakpoints conditions and counts can be added to any line, and the code run, traced or stepped.

If the PC of the target machine is pointing at a line in the current disassembly display, it is indicated by a *greater than* sign (>). If a line contains a breakpoint, that line is displayed in a different colour; the breakpoint count and expression details are shown at the end of the line.

The **Hex Window** displays memory in hexadecimal, either in byte, word or long word form. Like the Register Window, contents can be changed:

- by typing directly at the current cursor location;
- entering an expression for evaluation, by pressing the ENTER key to display an expression input window;
- pressing + or - will increment or decrement the value at the cursor position.

The **Watch Window** allows variables, tables and code locations to be monitored during running, by displaying specific memory addresses. These addresses are normally determined by expressions, including symbolic names, entered via the Watch Window.

The **Text** or **File Window** allows a text file to be viewed directly. Note that pressing ENTER, while the cursor is in the File Window, allows entry of a further file name for display.

The **Source Level Window** is an extension of the File window. Source Level mode is entered by pressing TAB, to display the current PC location, in a File window. It may be an empty File window, obtained by pressing ENTER in response to the filename prompt. This will load the appropriate source file into the Text window and position the cursor on the PC line. Note that, when in Source mode, line numbers are added to the left side of the window display and the PC line is indicated with a '>' after the line number, similar to a Disassembly window.

The programmer can step, trace, and set breakpoints in the source code in much the same way as a Disassembly window. The cursor in the currently active text window will track the PC during a trace. Note, however, that unlike tracing in a Disassembly window, a trace at Source Level may trace more than one instruction. Rather, it will trace the entire source *line*, which, if it is a macro or a 'C' source line, may correspond to the execution of one or more instructions. Similarly F8 (Stepover) will step-over the entire source line, which could be equivalent to stepping over several subroutine calls.

If the programmer is unsure of how a Source Level operation will behave, a Disassembly window can be viewed at the same time, to determine how the operations correspond to actual processor instructions. As the cursor tracks the PC at each step or stepover operation, if the PC should enter a region of the target code for which there is no Source Level information available, the window display will switch to Disassembly mode. The trace can be continued, and when the PC returns to a region for which there is Source information, the window will switch back to the Text display.

In order to use any of the Source Level features you must have extra debugging information in your symbol file(s). This information is added to the symbol files by the Assembler if the **/zd** switch was specified on the command line.

Using Debugger Windows

The following key strokes and mouse actions allow the programmer to exercise control over the Debugger display - note that a complete list of all key and menu options is given later in the chapter.

Moving between windows

Use one of the following methods to move between Debugger windows:

- Press F1, followed by an *up* or *down* cursor key to point to the required window;
- Press Shift, plus *up* or *down* cursor key;
- Point at the required window with the mouse and click.

To select the Window type

To change the type of the currently selected window:

- Either use the mouse to select the SET TYPE option from the WINDOW menu;
- Or press Shift and F1;
- In each case, a selection window is presented - use the mouse, or the cursor keys plus ENTER, to choose the new type.

Re-sizing Windows

To change the size of a Debugger window:

- Position the cursor in the required window;
- Press F2;
- Use the *up* or *down* cursor key to move the selected window edge to the desired size;
- Press ENTER to confirm.

Note that the currently selected window may be zoomed to fill the screen by pressing Control-Z; press again to re-present the original display.

To split an existing window

To add another window to the display:

- Position the cursor in the required window;
- Press F3;
- The new window is the same type as the source window.

Joining two windows

To remove a Debugger window:

- Position the cursor in the required window;
- Press F4;

- Use the *up* or *down* cursor key to select the window edge to be removed;

- Press ENTER to confirm.

To move the cursor within a window

The cursor control keys allow the re-positioning of the cursor in the selected window, as follows:

- Register Window - Use the four arrow keys to move between register values; the HOME key positions the cursor in the top left register field.

- Watch Window - Use the four arrow keys to move between adjacent lines and characters; the HOME key positions the cursor in the top left character position.

- Disassembly and Text Window - Use the up and down arrows to move the highlight bar; the HOME key moves the line under the cursor to the top of the window.

- Hex Window - Use the four arrow keys to move between adjacent lines and bytes/words; the HOME key moves the byte/word under the cursor to the top left of the window.

Locking a window

A window can be locked into displaying a specific memory region, as follows:

- Pressing Alt-L, and entering an address, or an expression which evaluates to an address, in the input box;

- Selecting the LOCK option from the WINDOW menu;
- Pressing Control-L turns the lock on and off.

A display can be locked to the expression **&0**; this allows the Debugger to be started with a window pointing to an address or label specified on the command line - see page 158.

If a lock expression is set, but de-activated by Control-L, the Debugger will start-up with the display initially positioned at the lock address, but the window start can subsequently be changed with the cursor keys, etc., as normal.

General Mouse Usage

- Clicking the left mouse button re-positions the cursor to the site of the click. If the new position is in another window, it will become the active window.

- Clicking the right mouse button on a register in the Register window will open an expression input box.

- Clicking the right mouse button on a memory field in the Hex window will open an expression input box.

- Clicking the right mouse button on a line in the Disassembly window toggles a breakpoint.

- A window can be re-sized by clicking the left mouse button on a window edge and dragging it to the new position.

- Dragging a window border to the edge of the window deletes the window.

Keyboard Options

The following table is a complete list of keyboard options, categorised by function. Many of these functions are duplicated by Menu options; however, such functions are shown in both lists for reference purposes.

Expressions

At many points in the session, the Debugger will prompt for input - this can often take the form of an expression for evaluation. Expressions in the Debugger follow the same rules as the Assembler (see page 23 onwards), with the following exceptions:

- Expressions may contain processor registers.
- The Debugger assumes a radix of hexadecimal; to indicate a number is decimal, it is preceded by a # sign.
- Indirect addresses are indicated by square brackets [ ].
- Where appropriate, the Debugger assumes an indirect datum is a long word; this can be overriden by use of the @ operator, in place of the dot, together with *b* or *w*, following the square bracket.

Prompts

Each time that the Debugger requests input, the reply is stored. These stored prompts form a history, which can be accessed and edited at data entry time, by using the up and down arrow keys. Note that, when the Debugger closes, the last four historic entries in each class are stored on the configuration file, and restored the next time that the Debugger is loaded.

Key(s)	*Effect*
Leaving the Debugger	
Ctrl-X	Exit the Debugger, without saving the current configuration
Alt-X	Exit the Debugger and save the current configuration
Alt-Z	Exit to DOS shell; type EXIT to return to the Debugger
Window Handling	
F1	Move to next window
F2	Re-size Window
F3	Divide a Window
F4	Delete a Window
Shift-Arrows	Move to selected Window
Ctrl-Z	Zoom current Window; again to restore original display
Shift-F1	Select Window Type
Debug Control	
Ctrl-F2	Start paused Debugging Session, if using a CPE file
Esc	Halt the target machine at first opportunity
Shift-Esc	Halt the Target machine, turning off all interrupts
Alt-R	Restore Registers from previous Save
Alt-I	Set the update interval; the interval is input in 18ths of a second, therefore, 18 means once a second, 9 means twice a second, etc.
Alt-U	Turn update mode on or off

Key(s)	*Effect*
+	Increment Label Level
-	Decrement Label Level
F6	Run Target code until the instruction under the cursor is reached
F7	Trace Mode; traps and Line A calls are stepped over
F8	Stepover mode; subroutine calls and DBRA instructions are stepped over
F9	Run Target code from current Program Counter
Shift-F7	Trace traps and Line A calls
Shift-F9	Run to address, specified in input window
Alt-F4	Backtrace; this function provides an UNDO of the updates effected by the latest trace (the Debugger keeps a record of about 200 instructions, depending on their content). Note that updates to certain write registers in the target machine, and memory areas designated as write only, cannot be undone.

File Accessing

<	Upload specified data from the Target to a named file on the PC
>	Download a file to the Target
Shift-F10	Load a new configuration file
Alt-S	Send specified section of Disassembly to a PC file

Miscellaneous

F10	Select a Menu Option
Alt-H	Hex Calculator; enter an expression to be evaluated
Alt-D	Display the amount of free memory

Key(s)	*Effect*
Alt-*n*	Switch to Virtual Screen *n* (1 - 9 plus 0 = 10).
Disassembly Window	
Up/Down Arrows	Move highlight bar
Left/Right Arrows	Move display by one word
PgUP/Dn	Move display by a page
Home	Move display so that currently highlighted line is at the top
Alt-G	Go to address specified in input window
Tab	Move the highlight bar to the Program Counter
Shift-Tab	Make the Program Counter the same as the currently highlighted address
Alt-L	Lock the display to a specified address
Ctrl-L	Turn Lock on or off
Alt-M	Toggle default MX state, which is used to disassemble instructions for which the actual MX cannot be determined form the context or from MX records in the symbol table
Ctrl-N	Note MX state at cursor position; creates an MX record in the symbol table for the address at the cursor
ENTER	Mini-Assembler; displays an input box, to enter a single line of source code to be assembled and inserted at the location under the cursor
File and Source Windows	
Tab	Load appropriate Source file; when in Source Level mode, use keys as in Disassembly window

Key(s)	*Effect*
Breakpoints	
Alt-C	Enter condition for the highlighted breakpoint
Ctrl-C	Enter count for the highlighted breakpoint
F5	Turn highlighted breakpoint on or off
Shift-F5	Clear all current breakpoints
Shift-F6	Reset all current breakpoint counts
Hex Window	
Arrows	Move to adjacent byte/word/long word
PgUp/Dn	Move display by one page
Home	Move display so that currently highlighted byte/word/long word is at the top
Alt-W	Switch display between byte, word and long word
ENTER	Change contents of current location to the result of an expression; entered in an input window
0-9, A-F	Directly change contents of highlighted location
+	Increment contents of highlighted location
-	Decrement contents of highlighted location
Alt-G	Go to address specified in input window
Alt-F	Move display to address contained in highlighted location
Register Window	
Arrows	Move to next register
Home	Move to top left register

Key(s)	*Effect*
ENTER	Change contents of current register to the result of an expression; entered in an input window
0-9, A-F	Directly change contents of highlighted register

Watch Window

Arrows	Move to next watch expression
Home	Move to top watch expression
Ins	Add a new watch expression
Del	Delete the highlighted watch expression

Menu Options

The DBUG658 menu affords easy mouse access to the commonest Debugger functions. Note that, if no mouse is available, the Menu can still be accessed by pressing F10.

Option	*Effect*
FILE	
Reload	Reload the last executable file
Download	Download a file to the Target
Upload	Upload specified data from the Target to a named file on the PC
Disassemble	Send specified section of Disassembly to a named PC file
Exit to DOS	Exit to DOS shell; type EXIT to return to the Debugger
Exit Debugger	Exit the Debugger and save the current configuration
RUN	
Go	Run Target code from the current Program Counter
Stop	Halt the Target machine, turning off all interrupts
To Address	Run to address, specified in input window
Backtrace	This function provides an UNDO of the updates effected by the latest trace (the Debugger keeps a record of about 200 instructions, depending on their content). Note that updates to certain write registers in the target machine, and memory areas designated as write only, cannot be undone.

Option	*Effect*
WINDOW	
Set Type	Select Window Type
Lock	Lock the display to the address entered in an input window
Print	Output screen to system printer
Set Tabs	Enter up to 8 tab positions, in decimal, separated by spaces; note this function is only relevant to File and Source Disassembly windows
CONFIG	
Load	Load a new configuration file
Save	Save the current configuration to the specified file
CPU	
Save Regs	Save the current state of the registers
Reset Regs	Reload the previously saved register state
Reset	Reset the Target Processor
STEP	Trace Mode; traps and Line A calls are stepped over
STEPOVER	Stepover mode; subroutine calls and DBRA instructions are stepped over

The Link Software

The adapter hardware contains software in ROM to enable it to communicate with the host PC. In addition, this ROM contains code to perform some functions which may be useful in the development stages of a product - see page 213, Target Interface Software Functions. The target interface software hooks the following 65816 exception vectors by default:-

$00FFE4	COP	dw $FFF4
	(used for miscellaneous firmware functions)	
$00FFE6	BRK	dw $FFF0
	(used for breakpoints)	
$00FFE8	ABORT	dw $FFF0
$00FFEA	NMI	dw $FFF0
$00FFEC		dw $0000
$00FFEE	IRQ	dw $FFF0

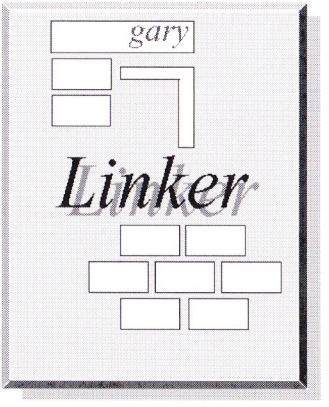
The memory at $00FFF0 is set-up to contain:-

$00FFF0	JML	BRKentry	(JML $7DE00B)
$00FFF4	JML	COPentry	(JML $7DE00E)
$00FFF8	JML	coldstart	(JML $7DE000)

In particular, the *BRK* and *COP* vectors are necessary for the correct function of the Debugger and the Line-A vector is used to provide various user functions. The application software will inevitably need to overwrite these vectors but care should be taken to initialise them as documented above - see also notes on COP $05 instruction, page 215.

CHAPTER 11

The PSYLINK Linker



The Psy-Q Linker, **PSYLINK**, is a fully-featured linker, which works with all processor types, and is compatible with other popular cross-compilers, such as Sierra and Aztec C. It facilitates the splitting of complex programs into separate, manageable modules, which can be recombined by PSYLINK into a final, single program.

This chapter discusses the linker, together with the Librarian utility, under the following headings:

- Command Line Syntax
- Linker Command Files
- XDEF, XREF and PUBLIC
- GLOBAL

The Linker-associated Assembler directives are repeated here for ease of reference.

Command Line Syntax

Description The PSYLINK link process is controlled by a series of parameters on the command line, and by the contents of a Linker command file. The syntax for the command line is as follows:

Syntax

PSYLINK */switches sourcefiles,outputfile,symbolfile,mapfile,libraries*

If a parameter is omitted, the separating comma must still appear, unless it is the last parameter of the line.

Linker Switches

Switches are preceded by a forward slash, and separated by commas. The following switches are available:

Switch	*Description*
/b	Specifies that the linker should run in 'big' mode. This allows the linker to link larger programs but with a link-time penalty.
/c	Tells the linker to link case sensitive; if it is omitted, all names are converted to upper case.
/d	Debug Mode - perform link only.
/e *symb=value*	Assigns *value* to *symbol.*
/i	Invokes a window containing Link details.
/m	Output all external symbols to the map file.

/o	*address*	Set an address for an ORG statement.
/o	*?address*	Request to target to assign memory for ORG.
/p		Create output as binary.
/r	*format*	Create machine specific relocatable output.
/u	*number*	Specify the unit number in a multi-processor target.
/x	*address*	Set address for the program to begin execution.

Sourcefile(s) A list of code source files, output by the ASM658 Assembler. File names are separated by spaces or plus (+) signs; if the file starts with an @ sign, it signifies the name of a Linker command file - see page 183 for a description of the format .

Outputfile The destination file for the output object code; if omitted, no object code is generated. If the output file name is in the format **T*n*:**, the object code is directly sent to the target machine - ***n*** specifies the SCSI device number.

Symbolfile The destination file for the symbol table information for use by the Debugger.

Mapfile The destination file for map information.

Libraries Library files available - see page 189 for further information.

Linker Command Files

Command files contain instructions for the Linker, about source files and how to organise them. The Linker command file syntax is much like the Assembler syntax, with the following commands available:

Commands

INCLUDE *filename*	Specify name of object file to be read.
INCLIB filename	Specify library file to use
ORG *address*	Specify ORG address for output
WORKSPACE *address*	Specify new target workspace address
name **EQU** *value*	Equate *name* to *value*
REGS *pc=address*	Set initial PC value
name **GROUP** *attributes*	Declare group
name **SECTION** *attributes*	Declare section with attributes
SECTION *name*[,*group*]	Declare section, and optionally specify its group
name **ALIAS** *oldname*	Specify an ALIAS for a symbol name

Also available,

PROC, ENDP, LABEL, CALL, JUMP	See page 91 onwards, for description

Group attributes:

BSS	- group is uninitialised data
ORG(*address*)	- specify group's org address
OBJ(*address*)	- specify group's obj address
OBJ( )	- group's obj address follows on from previous group
OVER(*group*)	- overlay specified group
FILE(*"filename"*)	- write group's contents to specified file
SIZE(*maxsize*)	- specify maximum allowable size

Remarks

- Sections within a group are in the order that section definitions are encountered in the command file or object/library files.
- Any sections that are not placed in a specified group will be grouped together at the beginning of the output.
- Groups are output in the order in which they are declared in the Linker command file or the order in which they are encountered in the object and library files.
- Sections which are declared with attributes, that is, not in a group, in either the object or library files, may be put into a specified group by the appropriate declaration in the Linker command file.

Examples

```
                include   "inp.obj"
                include   "sort.obj "
                include   "out.obj"

                org       1024
                regs      pc=progstart

lowgroup        group
codegroup       group
bssgroup        group     bss

                section   datal,lowgroup
                section   data2,lowgroup

                section   codel,codegroup
                section   code2,codegroup

                section   tables,bssgroup
                section   buffers,bssgroup
```

XDEF, XREF and PUBLIC

Description If several sub-programs are being linked, to refer to symbols in a sub-program which are defined in another sub-program, use **XDEF**, **XREF** and **PUBLIC**.

Syntax

XDEF *symbol*[,*symbol*]
XREF *symbol*[,*symbol*]

PUBLIC *on*
PUBLIC *off*

Remarks

- In the sub-program where symbols are initially defined, the **XDEF** directive is used to declare them as externals.
- In the sub-program which refers the symbols, the **XREF** directive is used to indicate that the symbols are in a another sub-program.
- The Assembler does not completely evaluate an expression containing an **XREF**ed symbol; however, resolution will be effected by the linker.
- The **PUBLIC** directive allows the programmer to declare a number of symbols as externals. With a parameter of ***on***, it tells the Assembler that all further symbols should be automatically **XDEF**ed, until a **PUBLIC** ***off*** is encountered.

Examples Sub-program A contains the following declarations:

```
        xdef    Scores,Scorers
        xref    PointsTable
        ...
```

The corresponding declarations in sub-program B are:

```
        xdef    PointsTable
        xref    Scores,Scorers
        ...
```

```
        public  on
Origin  =       Mainchar
Force   dw      speed*origin
Rebound dw      45*angle
        public  off
```

GLOBAL

Description The **GLOBAL** directive allows a symbol to be defined which will be treated as either an **XDEF** or an **XREF**. If a symbol is defined as **GLOBAL** and is later defined as a label, it will be treated as an **XDEF**. If the symbol is never defined, it will be treated as an **XREF**.

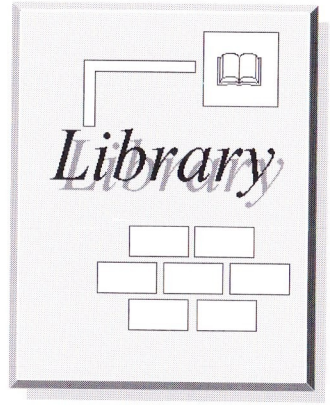
Syntax **GLOBAL** *symbol*[,*symbol*]

See Also **XREF**, **XDEF**, **PUBLIC**

Remarks This is useful in header files because it allows all separately assembled modules to share one header file, defining all global symbols. Any of these symbols later defined in a module will be **XDEF**ed, the others will be treated as **XREF**s.

CHAPTER 12

The Librarian



If the Linker cannot find a symbol in the object files, it can be instructed, by a Linker command line option, to search one or more object module Library files.

This chapter discusses Library usage and the **PSYLIB** library maintenance program:

- **PSYLIB** Command Line Syntax
- Using the Library feature

PSYLIB Command LIne Syntax

Description The Library program, **PSYLIB.EXE**, adds to, deletes from, lists and updates libraries of object module.

Syntax **PSYLIB** */switches library module...module*

where switches are preceded by a forward slash (/), and separated by commas.

See Also **PSYLINK**

Switches

/a	Add the specified modules to the library
/d	Delete the specified module from the library
/l	List the modules contained in the library
/u	Update the specified modules in the library
/x	Extract the specified modules from the library

Library The name of the file to contain the object module library.

Modules The object modules involved in the library maintenance.

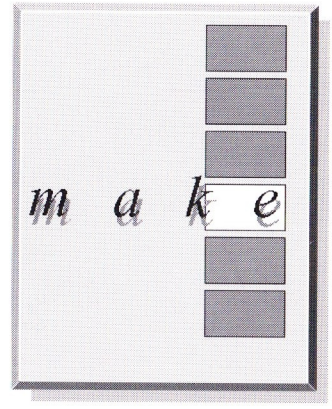
Using the Library feature

- To incorporate a Library at link time, specify a library file on the Linker command line - see page 182

- If the Linker locates the required external symbol in a nominated library file, the module is extracted and linked with the object code output by the Assembler

CHAPTER 13

The PSYMAKE Utility



PSYMAKE is a make utility for MS-DOS which automates the building and rebuilding of computer programs. It is general purpose and not limited to use with the **Psy-Q** system. The utility is discussed under the following headings:

- Command Line Syntax
- Format of the Makefile

PSYMAKE Command Line Syntax

Description **PSYMAKE** rebuilds only those components of a system that need rebuilding. Whether a program needs rebuilding is determined by the file date stamps of the target file and the source files that it depends on. Generally, if any of the source files are newer than the target file, the target file will be rebuilt.

Syntax **PSYMAKE** *[options] [target]*

Remarks

- Valid options are :

/b	Build all, ignoring dates
/d *name=string*	Define name as string
/f *filename*	Specify the MAKE file
/i	Always ignore error status
/q	Quiet mode; do not print commands before executing them
/x	Do not execute commands - just print them

- If no **/f** option is specified, the default makefile is MAKEFILE.MAK; if no extension is specified on the makefile name, .MAK will be assumed.

- If no *target* is specified, the first target defined in the makefile will be built.

Contents of the Makefile

The Makefile consists of a series of commands, governed by explicit rules, known as dependencies, and implicit rules. When a target file needs to be built, PSYMAKE will first search for a dependency rule for that specific file. If none can be found, PSYMAKE will use an implicit rule to build the target file.

Dependencies:

A dependency is constructed as follows :

```
targetfile : [sourcefiles]
    [command
        ...
     command ]
```

- The first line instructs PSYMAKE that the file *"targetfile"* depends on the files listed as *"sourcefiles"*.
- If any of the source files are dated later than the target file, or the target file does not exist, PSYMAKE will issue the commands that follow in order to rebuild the target file.
- If no source files are specified, the target file will always be rebuilt.
- If any of the source files do not exist, PSYMAKE will attempt to build them first, before issuing the commands to build the current target file. If PSYMAKE cannot find

any rules defining how to build a required file, it will stop and report an error.

- The target file name must start in the left hand column. The commands to be executed in order to build the target must all be preceded by white space (either space or tab characters). The list of commands ends at the next line encountered with a character in the leftmost column.

Examples

```
main.cpe: main.658 inc1.h inc2.h
    ASM658 main,main
```

This tells PSYMAKE that *main.cpe* depends on the files *main.658, inc1.h* and *inc2.h*. If any of these files are dated later than *main.cpe*, or *main.cpe* does not exist, the command *"ASM658 main,main"* will be executed in order to create or update *main.cpe*.

```
main.cpe: main.658 inc1.h inc2.h
    ASM658 /l main,main,main
    psylink main,main
```

Here, two commands are required in order to rebuild *main.cpe*.

Implicit Rules

If no commands are specified, **PSYMAKE** will search for an implicit rule to determine how to build the target file. An implicit rule is a general rule stating how to derive files of one type from another type; for instance, how to convert .ASM files into .EXE files.

Implicit rules take the form:

```
.<target extension>.<source extension>:
    command
    [...
    command ]
```

- Each *<extension>* is a 1, 2 or 3 character sequence specifying the DOS file extension for a particular class of files.

- At least one command must be specified.

Examples

```
.bin.658:
    asm658 /p $*,$*
```

This states that to create a file of type *.bin* from a file of type *.658*, the ASM658 command should be executed. (See below for an explanation of the $* substitutions.)

Executing commands :

Once the commands to execute have been determined, PSYMAKE will search for and invoke the command. Search order is:

- *current directory;*
- *directories in the path.*

If the command cannot be found as *A*.EXE or *A*.COM file or the command is *A*.BAT file, PSYMAKE will invoke COMMAND.COM to execute the command/batch file. This enables commands like CD and DEL to be used.

Command prefixes :

The commands in a dependency or implicit rule command list may optionally be prefixed with the following qualifiers :

@	- suppress printing of command before execution
- *number*	- abort if exit status exceeds specified level
-	- (without number) ignore exit status (never abort)

- Normally, unless **/q** is specified on the command line, PSYMAKE will print a command before executing it. If the command is prefixed by @, it will not be printed.

- If a command is prefixed with a hyphen, followed by a number, PSYMAKE will abort if the command returns an error code greater than the specified number.

- If a command is prefixed with a hyphen without a number, PSYMAKE will not abort if the command returns an error code.

- If neither a hyphen or a hyphen+number is specified, and **/i** is not specified on the command line, PSYMAKE will abort if the command returns an error code other than 0.

Macros

A macro is a symbolic name which is equated to a piece of text. A reference to that name can then be made and

will be expanded to the assigned text. Macros take the form:

```
name = text
```

- The text of the macro starts at the first non-blank character after the equals sign (=), and ends at the end of the line.
- Case is significant in macro names.
- Macro names may be redefined at any point.
- If a macro definition refers to another macro, expansion takes place at time of usage.
- A macro used in a rule is expanded immediately.

Examples

```
FLAGS = /p /s
...
.bin.658:
    ASM658 $(FLAGS) /p $*,$*
```

The *$(FLAGS)* in the ASM658 command will be replaced with */p /s*.

Pre-defined macros :

The following pre-defined macros all begin with a dollar sign and are intended to aid file usage:

$d Defined Test Macro, e.g.:

```
!if $d(MODEL)
# if MODEL is defined ...
```

$*	Base file name with path, e.g.: C:\PSYQ\TEST
$<	full file name with path, e.g.: C:\PSYQ\TEST.658
$:	path only, e.g.: C:\PSYQ
$.	full file name, no path, e.g.: TEST.658
$&	base file name, no path, e.g.: TEST

- The filename pre-defined macros can only be used in command lists of dependency and implicit rules.

Directives :

The following directives are available:

!if *expression*
!elseif *expression*
!else
!endif

These directives allow conditional processing of the text between the *if*, *elseif*, *else* and *endif*. Any non-zero expression is TRUE; zero is FALSE.

!error *message*	Print the message and stop.
!undef *macroname*	Undefines a macro name.

Expressions :

Expressions are evaluated to 32 bits, and consist of the following components :

Decimal Constants	e.g. 1 10 1234
Hexadecimal	e.g. $FF00 $123abc
Monadics	- ~ !
Dyadics	+ - * / % > < &
	| ^ && ||
	> < >= <= == (or =)
	!= (or <>)

- The operators have the same meanings as they do in the C language, except for = and <> ,which have been added for convenience.

Value assignment :

Macro names can be assigned a calculated value; for instance:

```
NUMFILES == $(NUMFILES)+1
```

(Note two equals signs in value assignment)

This evaluates the right hand side, converts it to a decimal ascii string and assigns the result to the name on the left.

In the above example, if NUMFILES was currently "42", it will now be "43".

Note that:

```
NUMFILE = $(NUMFILES)+1
```

would have resulted in NUMFILES becoming "42+1".

- Undefined macro names convert to '0' in expressions and null string elsewhere.

Comments:

Comments are introduced by a hash mark (#):

```
main.exe: main.asm          # main.exe only depends
                            # on main.asm

# whole line comment
```

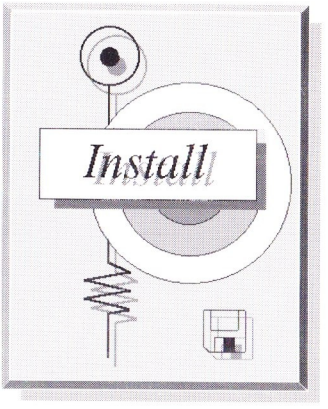
Line continuation:

A command too long to fit on one line may be continued on the next by making '\' the last character on the line, with no following spaces/tabs:

```
main.exe : main.asm  i1.h i2.h \
           i3.h i4.h
```

CHAPTER 14

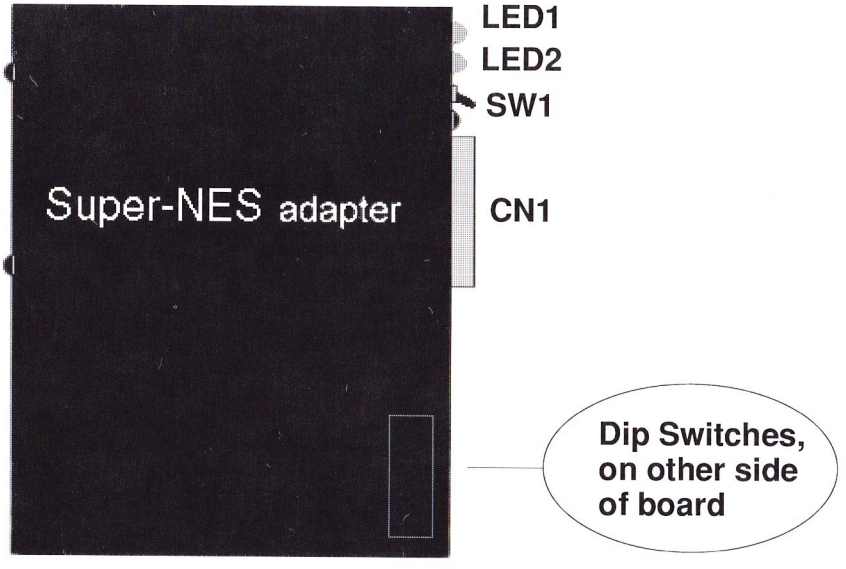
Setting up the Target machine



Fitting and configuring the Target Interface Adapter to the Super-NES is a simple task - this is discussed in the first part of this chapter. Also described are useful functions in the adapter firmware, and some fileserver routines specific to the Super-NES, as follows:

- Installation of the Target Interface Adapter
- Firmware diagnostics
- Target Interface Software Functions
- Fileserver Functions

Installing the Target Interface Adapter



Installation Install the target interface adapter as follows:

- The adapter is inserted into the Super-NES cartridge slot with the curved side facing towards the front on the unit.
- An appropriate Super-Nintendo cartridge, of the same country of origin as the console, **must** be plugged into the through-connector, for the Super-NES adapter interface to function. This not only supplies the security chip, but can also provide additional facilities, such as DSP chips and battery-backed RAM.
- The cable from the host PC adapter board is plugged into the adapter at CNl.

- The DIP switches should be set before the adapter is plugged in, since they are inaccessible once the board has been fitted.

- SW1 toggle switch is initially in the middle position.

- LED1 comes on and stays on; LED2 will light briefly and then go off, when the adapter completes its self-test.

DIP Switch settings:-

Note that Dip Switches must be set before the adapter interface is fitted.

Switch No	*Description*
1, 2, 3	Select SCSI ID to be used by adapter; default is OFF, OFF, OFF for ID 0.
4	ON to enable fast hardware mapping. This replaces 64K of RAM from the fast area ($FDXXXX), which may be undesirable if developing a 16 Megabit fast mode product. Otherwise, set this switch ON, for maximum SCSI link performance.
5,6	Cartridge type to emulate - see table below
7	Firmware bank; this switch should be ON
8	Not used

Settings of switches 5 and 6, for the cartridge emulation type, are as follows:

SW5	*SW 6*	*Cartridge type*
OFF	OFF	Standard mode 20 - 1 Megabyte slow. RAM in top 32K banks $00 to $1F, RAM in top 32K banks $80 to $9F (fast).
ON	OFF	Extended mode 20 - 2 Megabyte slow. RAM in top 32K banks $00 to $3F, RAM in top 32K banks $80 to $BF (fast).
OFF	ON	Standard mode 21 - 4 Megabyte fast. RAM in all 64K banks, $C0 to $FF. Top 32K of $C0 to $FF shadowed in top 32K of banks $00 to $3F and $80 to $BF.
ON	ON	Extended mode 21. 4 Megabyte fast or slow. RAM in all 64K banks, $C0 to $FF (fast) also shadowed banks $40 to $7C (slow). Top 32K of these banks is also shadowed in top 32K of banks $00 to $3F and $80 to $BF.

Unless emulating a particular cartridge type, the most usual mode is with both switches ON. This maps RAM into all of the available cartridge address space, although the RAM is shadowed to bank $00 to allow the cartridge to respond to CPU reset. This will suit the majority of developers who are not utilising any special hardware on the cartridge, such as DSP chips. Any areas that are not mapped with RAM, are shadowed from the cartridge plugged into the through-connector. This allows the

programmer to make use of hardware such as battery backed RAM, or DSP chips.

Toggle Switch

SWI is a 3 position toggle switch, which controls whether the cartridge ROM area is mapped to the cartridge on the through-connector, or to the emulation RAM on the adapter, as well as what happens when the reset button is pressed. It may be set as follows:

UP	The ROM area is the cartridge on through-connector; resets to adapter firmware.
CENTRE	The ROM area is the emulation RAM; resets to adapter firmware.
DOWN	The ROM area is the emulation RAM; resets to emulation RAM, that is, the program's startup code.

Note that the target machine can be powered-on with the switch in the CENTRE position, then code downloaded and the switch set to the DOWN position. Thereafter, pressing the RESET button will reset to the program code in emulation RAM. It also resets the sound CPU, so that the application can verified from a full reset condition.

LED's

LED1 lights at power-up or reset, and stays on to show that the emulation RAM is enabled.

LED2 comes on briefly at power-up or reset, then goes off when the adapter completes its self-test. If the unit fails its initial self test, LED2 will flash on and off to indicate the error. Otherwise, if no error is detected, LED2 lights while the emulation RAM is write protected.

Additional Development Considerations

The adapter hardware makes use of the 64K memory range at address $7DXXXX. This area is, therefore, unavailable to any application program. The **ASM658.EXE** Assembler gives no indication that the program is attempting to access illegal memory - it is the programmer's responsibility to ensure that no code is ORGed into this area. Note also that **DBUG658.EXE** functions will not attempt to access this area.

Similarly, this memory range must not be accessed if the adapter hardware is run in fast mode, with Dip switch 4 set to ON. If the program requires less than 4 Megabytes of memory, the lost 64K should not present a problem; however, if the program requires to access the full 16 Megabits of memory, from $C00000 to $FFFFFF, be sure that Dip-switch 4 is OFF. This will only slightly reduce the performance of the adapter SCSI comms link.

Adapter Firmware Diagnostics

- At power up or reset, if the adapter firmware is enabled, and the toggle switch is in the *up* or *middle* positions, the firmware ROM will take control of the machine and run a few self tests on the adapter hardware.

- If no errors are detected, and if the cartridge emulation RAM is enabled, that is, the toggle switch is in the *middle* position, then certain interrupt vectors will be pointed into the adapter's firmware; the adapter will await commands from the host PC.

- If the joypad START button is held down at power-on or reset, the firmware will perform more thorough diagnostic tests and report on the currently configured state of the adapter, showing the results on the console video display.

Target Interface Software Functions

The adapter firmware provides a few useful functions which can be accessed via the COP instruction. COP $XX instructions can be entered in the Assembler source code; the $XX selects the function, and additional parameters are passed in registers.

These functions allow software to interact with, and change the configuration of, the adapter downloader firmware. In most cases, the default configuration will be adequate for the programmer's needs, with function usage centring around the Poll call (COP $00).

COP Instructions

Use of unsupported COP functions will report a normal BRK interrupt to the PC Debugger. The following functions are currently supported:

COP $00 Poll the Host PC

The source code should periodically make one of these calls, to allow the PC to access the host memory while the program is running. This will, for example, allow the Debugger to edit memory, stop the target program, and to step, trace and insert breakpoints.

A suitable Poll rate might be 25 to 100 times a second. Typically, the poll call is placed in the program's main loop, or in the VBL interrupt handler.

The Poll call takes very little time, if the PC is not requesting or sending any data. If the PC has requested an access to the target memory, the call takes a little longer.

In this context, see also page 216; the **COP $07** instruction can be used to enable or disable interrupts, while the poll call is transferring data.

COP $01 Cold Start the adapter firmware.

All internal flags and variables are reset to their power-up state.

COP $02 Warm Start the adapter firmware.

This execution of the program is halted, at that point. Control is returned to the adapter downloader firmware, which awaits commands from the host PC.

COP $03 Soft Re-Entry to Downloader program.

This is almost identical to the above function, except that the PC is advanced past this instruction before halting. This allows the insertion of temporary pause points so that results can be examined in the Debugger. Pressing the run button allows the program to continue from that point.

COP $04 Get Host Status.

Returns NE & CS flags (Z flag =0, C flag=1), if PC is waiting to access the target; that is, if a COP $00 Poll call would transfer data. Or it returns EQ & CC flags (Z flag = 1, C flag=0), if the PC does not wish to access the target.

COP $05 Reset Interrupt Vectors.

This call re-installs the interrupt vectors, as hooked by the adapter firmware at power-up or reset. By default, the adapter hooks into the following vectors:-

```
$00FFE4     COP     dw $FFF4
                    ;used for misc firmware functions
$00FFE6     BRK     dw $FFF0
                    ;used for breakpoints
$00FFE8     ABORT   dw $FFF0
$00FFEA     NMI     dw $FFF0
$00FFEC             dw $0000
$00FFEE     IRQ     dw $FFF0
```

The memory at $00FFF0 is set-up to contain:-

```
$00FFF0     JML     BRKentry    (JML $7DE00B)
$00FFF4     JML     COPentry    (JML $7DE00E)
$00FFF8     JML     coldstart   (JML $7DE000)
```

Note that, if you change the COP or BRK vectors, the operation of the Debugger may be interfered with; the other vectors are merely pointed to the standard BRK handler. However, it is probable that the NMI and IRQ will be vectored to the application's own handler. Since these vectors are in cartridge space, normally ROM, they cannot be written to at runtime. The final program image

may overwrite all these vectors, but, to be able to use the Debugger in the program, the COP and BRK vectors, as well as JMLs at $FFF0, should be set up as above.

COP $06 NOP

This instruction does nothing, except it always returns flags EQ/CC (no error). The op-code has been intentionally left blank, for consistency with 68K development systems.

COP $07 Set Interrupt status on entry to downloader.

Passed: Acc.b Accumulator LSB bits 0 and 1, used to flag whether interrupts are to be disabled during firmware operations.

Bit 0 - clear to disable IRQ during firmware BRK handler; set to leave I flag unchanged.

Bit 1 - clear to disable IRQ during firmware POLL call; set to leave I flag unchanged.

This call can be used to enable IRQ during a poll call. Typical uses are to avoid missing interrupts while the Debugger is accessing target memory, or has stopped the target. The value passed in bit 0 sets the interrupts which are allowed to run, after a breakpoint is triggered on the target. The default is for the I flag to be left unchanged, the same as before the COP $00 poll call or BRK instruction.

COP $08 Set Write-Protect status of emulation RAM.

Passed: Acc.b = $FF to write protect cartridge emulation RAM
= $00 to write enable cartridge emulation RAM

Normally, the cartridge emulation RAM is write protected while the application software is running, except during interrupt handlers that are triggered while a poll call is in operation. This is to emulate a cartridge, which is ROM and cannot be written. For test purposes, however, the cartridge emulation RAM can be left write enabled. However, it should be remembered that code which writes to this RAM will not function once it has been installed in a ROM cartridge.

COP $09 Set target (host) ID for fileserver functions

Passed: Acc.b = ID of host to be used by fileserver functions (0 to 7 or -1)

If this value has not been specified, by default the fileserver will use the ID of the last device that connected to it. If the target has not been connected, the default ID will be 7, same as the default ID of a PC board.
Setting the ID to -1 causes the fileserver to use the ID of the last host that connected to it, or 7 if no connection has yet been made. Values higher than 7, or the same as the target SCSI ID, return an error status, processor flags = NE & CS.

Fileserver Functions

The target adapter also contains software to provide fileserver functions. As with the **Psy-Q** target BIOS calls, these are accessed via the COP $XX instruction.

Note that all fileserver functions return CC & EQ status in processor flags, if no error is encountered; otherwise CS & NE are returned, with an error code in the Accumulator. Although this error code is an 8 bit value, it is sign-extended so that it can be checked in either 8 or 16 bit mode. The M,X,and I bits of the status register are returned unchanged, although all fileserver functions are designed to be called with all registers in 16 bit mode, that is, M and X bits = 1. Interrupts are disabled during the fileserver call.

Unless otherwise documented, all fileserver functions operate as per the corresponding DOS file functions. The names given to the functions below are the names which will be displayed by the Psy-Q Super-NES Debugger, in a disassembly window.

COP $40 fsINIT - initialise remote filing system

Passed: Nothing

Returns: CC/CS. Acc.b = standard error status

This function resets the disk system of the host, ready for a new session; all remote-opened files are closed.

COP $41 fsCREATE - create a file

Passed:	DB:X.w	Pointer to zero-terminated filename
	DB:Y.w	File attributes
Returns:	X.w	File handle

COP $42 fsOPEN - open a file

Passed:	DB:X.w	Pointer to zero terminated filename
	DB:Y.w	Open mode
Returns:	X.w	File handle

COP $43 fsCLOSE - close a file

Passed:	Acc.w	File handle
Returns:	Nothing but standard error status	

COP $44 fsREAD - read bytes from a file

Passed:	Acc.w	File handle
	DB:X.w	Address of destination for data
	DB:Y.w	Number of bytes to be read
Returns:	Y.w	Number of bytes actually read

COP $45 fsWRITE - write bytes to a file

Passed:	Acc.w	File handle
	DB:X.w	Address to write data from
	DB:Y.w	Number of bytes to be written
Returns:	Y.w	No of bytes actually written

COP $46 fsSEEK0 - seek to absolute file position

Passed:	Acc.w	File handle
	X.w	LSW offset
	Y.w	MSW offset of new file pointer
Returns:	X,Y	New value of file pointer

COP $47 fsSEEK1 - seek relative to current file pointer

Passed:	Acc.w	File handle
	X.w	LSW offset
	Y.w	MSW offset of new file pointer
Returns:	X,Y	New value of file pointer

COP $48 fsSEEK2 - seek to *n* bytes before end of file

Passed:	Acc.w	File handle
	X.w	LSW offset
	Y.w	MSW offset of new file pointer
Returns:	X,Y	New value of file pointer

APPENDICES

- Appendix A - Errors during Assembly
- Appendix B - Errors during Linking
- Appendix C - Librarian Errors
- Appendix D - The SPC700 Assembler

Appendix A - ASM658 Error Messages

Format: In the list below, *%x* represents the variable part of the error message, as follows:

%s is replaced by a string
%c is replaced by a single character
%d is replaced by a 16 bit decimal number
%l is replaced by a 32 bit decimal number
%h is replaced by a 16 bit hexadecimal number
%n is replaced by a symbol name
%t is replaced by a symbol type, e.g. section, symbol or group.

Assembler Messages:

'%n' cannot be used in an expression
%n will be the name of something like a macro or register

'%n' is not a group
Group name required

'%n' is not a section
Section name expected but name %n was found

Alignment cannot be guaranteed
Warning of attempt to align that cannot be guaranteed due to the base alignment of the current section

Alignment's parameter must be a defined name
In call to alignment( ) function

Assembly failed
Text of the FAIL statement

Branch (%l bytes) is out of range
Branch too far

Cannot POPP to a local label
E.g. POPP @x

Cannot purge - name was never defined

Case choice expression cannot be evaluated
On case statement

Code generated before first section directive
Code generating statements appeared before first section directive

Could not evaluate XDEF'd symbol
XDEF'd symbol was equated to something that could not be evaluated

Could not open file '%s'

Datasize has not been specified
Must have a DATASIZE before DATA statement

Datasize value must be in range 1 to 256
DATASIZE statement

Decimal number illegal in this radix
Specified decimal digit not legal in current radix

DEF's parameter must be a name
Error in DEF( ) function reference

Division by zero

End of file reached without completion of %s construct
E.g. REPT with no ENDR

ENDM is illegal outside a macro definition

Error closing file
DOS close file call returned an error status

Error creating output file
Could not open the output file

Error creating temporary file
Could not create specified temporary file

Error in assembler options

Error in expression
Similar to syntax error

Error in floating point number
In IEEE32 / IEEE64 statement

Error opening list file
DOS open returned an rror status

Error reading file
DOS read call returned an error status

Error writing list file
DOS write returned an error status or disk full

Error writing object file
DOS write call returned an error or disk is full

Error writing temporary file
Disk write error, probably disk full

Errors during pass 1 - pass 2 aborted
If pass 1 has errors then pass 2 is not performed

Expanded input line too long
After string equate replacement, etc. line must be <= 1024 chars

Expected comma after '
'...' bracketed parameter in MACRO call parameter list

Expected comma after operand

Expected comma between operands

Expected comma between options
In an OPT statement

Expecting '%s' at this point
Expecting one of ENDIF/ENDCASE etc. but found another directive

Expecting '+' or '-' after list command
In a LIST statement

Expecting '+' or '-' after option
In an OPT statement

Expecting a number after /b option
On Command line

Expecting comma between operands in INSTR

Expecting comma between operands in SUBSTR

Expecting comma or end of line after list
In { ... } list

Expecting ON or OFF after directive
In PUBLIC statement

Expecting options after /O
On Command line

Expecting quoted string as operand

Expression must evaluate
Must be evaluated now, not on pass 2

Fatal error - macro exited with unterminated %s loop
End of macro with unterminated WHILE/REPT/DO loop. Due to the way the assembler works, this must be treated as a fatal error

Fatal error - stack underflow - PANIC
Assembler internal error - should never occur!

File name must be quoted

Files may only be specified when producing CPE or pure binary output
In FILE attribute of group

Forward reference to redefinable symbol
Warning that a forward reference was made to a symbol that was given a number of values in SET or = statements. The value used in the forward reference was the last value the symbol was set to.

Function only available when using sections

Group '%n' is too large (%l bytes)
Group exceeds value in SIZE attribute

GROUP's parameter must be a defined name
In GROUP( ) function call

GROUPEND's parameter must be a group name
Error in call to GROUPEND( ) function

GROUPORG's parameter must be a group
In call to GROUPORG( ) function

GROUPSIZE's parameter must be a group name
Error in call to GROUPSIZE( ) function

IF does not have matching ENDIF/ENDC

Illegal addressing mode
Addressing mode not allowed for current op code

Illegal character '%c' (%d) in input
Strange (e.g. control) character in input file

Illegal character '%c' in opcode field

Illegal digit in suffixed binary number
In alternate number form 101b

Illegal digit in suffixed decimal number
In alternate number form 123d

Illegal digit in suffixed hexadecimal number
In alternate number form 1abh

Illegal group name

Illegal index value in SUBSTR

Illegal label
Label in left hand column starts with illegal character

Illegal name for macro parameter
In macro definition

Illegal name in command
Target name in ALIAS statement

Illegal name in locals list
In LOCAL statement

Illegal name in XDEF/XREF list

Illegal parameter number
Maximum of 32 parameters

Illegal section name

Illegal start position/length in INCBIN

Illegal use of register equate
E.g. using a register equate in an expression

Illegal value (%l)

Illegal value (%l) for boundary in CNOP

Illegal value (%l) for offset in CNOP

Illegal value for base in INSTR

Initialised data in BSS section
BSS sections must be uninitialised

Label '%n' multiply defined

LOCAL can only be used inside a macro
LOCAL statement found outside macro

Local labels may not be strings
@x EQUS ... is illegal

Local symbols cannot be XDEF'd/XREF'd

MEXIT illegal outside of macros

Missing '(' in function call

Missing ')' after function parameter(s)

Missing ')' after file name
In FILE attribute

Missing closing bracket in expression

Missing comma in list of case options
In =... case selector

Missing comma in XDEF/XREF list

MODULE has no corresponding MODEND

Module may not end until macro/loop expansion is complete
If a loop / macro call starts inside a module then there must not be a MODEND until the loop / macro call finishes

Module must end before end of macro/loop expansion - MODEND inserted
A module started inside a loop / macro call must end before the loop / macro call does

More than one label specified
Only one label per line (can occur when second label does not start in left column but ends in ':')

Move workspace command can only be used when downloading
In WORKSPACE statement

Names declared with local must not start with '%c'
In LOCAL statement

NARG can only be used inside a macro
Use of NARG outside macro

NARG's parameter must be a number or a macro parameter name
Illegal operand for NARG( ) function

No closing quote on string

No corresponding IF
ENDIF/ELSE without IF

No corresponding DO
UNTIL without DO

No corresponding REPT
ENDR without REPT

No corresponding WHILE
ENDW without WHILE

No matching CASE statement for ENDCASE
ENDCASE without CASE

No source file specified
No source file on command line

Non-binary character following %

Non-hexadecimal character '%c' encountered
In HEX statement

Non-hexadecimal character starting number
Expecting 0-9 or A-F after $

Non-numeric value in DATA statement

OBJ cannot be specified when producing linkable output
OBJ attribute on group

Odd number of nibbles specified
In HEX statement

OFFSET's parameter must be a defined name
Error in OFFSET( ) function call

Old version of %n cannot be purged
Only macros can be purged

One string equate can only be equated to another
Attempt to equate to expression, etc.

Only one of /p and /l may be specified
On Command line

Only one ORG may be specified before SECTION directive

Op-code not recognised

Option stack is empty
POPO without PUSHO

Options /l and /p not available when downloading to target
On Command line

ORG ? can only be used when downloading output

ORG address cannot be specified when producing linkable output
No ORG group attributes when producing linkable output

ORG cannot be used after SECTION directive

ORG cannot be used when producing linkable output

ORG must be specified before first section directive
When using sections only one ORG statement may appear before all section statements (other than as group attributes)

Out of memory, Assembler aborting

Out of stack space, possibly due to recursive equates
Assemblers stack is full, possible cause is recursive equates, e.g. *x equ y+1 , y equ x*2*

Overflow in DATA value
DATA value too big

Overlay cannot be specified when producing linkable output
No OVER group attributes when producing linkable output

Overlay must specify a previously defined group name
Error in OVER group attribute

Parameter stack is empty
POPP encountered but nothing to pop

POPP must specify a string or undefined name

Possible infinite loop in string substitution
E.g. reference to x where x is defined as *x equs x+1*

Previous group was not OBJ'd
OBJ( ) attribute specified but previous group had no obj attribute to follow on from

Psy-Q needs DOS version 3.1 or later

Purge must specify a macro name

Radix must be in range 2 to 16

REF's parameter must be a name
Error in REF( ) function reference

Register not recognised
Expecting a register name but did not recognise

Remainder by zero
As for division by 0 but for % (remainder)

Repeat count must not be negative
REPT statement error

Replicated text too big
Text being replicated in a loop must be buffered in memory but this loop was too big to fit

Resident SCSI drivers not present
PSYBIOS does not appear to be loaded

SCSI card not present - assembly aborted

SECT's parameter must be a defined name
Error in SECT( ) function call

SECTEND's parameter must be a section name
Error in call to SECTEND( ) function

Section stack is empty
POPS without PUSHS

Section was previously in a different group
Section assigned to a different group on second invocation

SECTSIZE's parameter must be a section name
Error in call to SECTSIZE( ) function

Seek in output file failed
DOS seek call returned error status

Severity value must be in range 0 to 3
In INFORM statement

SHIFT can only be used inside a macro
SHIFT statement outside macro

Short macro calls in loops/macros must be defined before loop/macro

Short macros may not contain labels

Size cannot be specified when producing linkable output
SIZE attribute on group

Size specified in /b option must be in range 2 to 64
On command line

Square root of negative number

Statement must have a label
No label on, for example, EQU op

STRCMP requires constant strings as parameters

String '%n' cannot be shifted
String specified in SHIFT statement is not a multi-element string (i.e. {...} bracketed) and so cannot be shifted.

STRLEN's operand must be a quoted string

Symbol '%n' cannot be XDEF'd/XREF'd

Symbol '%n' is already XDEF'd/XREF'd

Symbol '%n' not defined in this module
Undefined name encountered

Syntax error in expression

Timed out sending data to target
Target did not respond

Too many characters in character constant
Character constants can be from 1 to 4 characters

Too many different sections
There is a maximum of 256 sections

Too many file names specified
On command line

Too many INCLUDE files
Limit of 512 INCLUDE files

Too many INCLUDE paths specified
Too many INCLUDE paths in /j options on command line

Too many output files specified
Maximum of 256 output files

Too many parameters in macro call
Maximum number of parameters exceeded

Too much temporary data
Assembler limit of 16m bytes of temporary data reached

TYPE's parameter must be a name
Call of TYPE( ) function

Unable to open command file
From Command line

Undefined name in command
Target name in ALIAS statement

Unexpected case option outside CASE statement
Found =... statement outside CASE/ENDCASE block

Unexpected characters at end of Command line

Unexpected characters at end of line
End of line expected but there were more characters encountered (other than comments)

Unexpected end of line
Line ended but more input was expected

Unexpected end of line in macro parameter

Unexpected end of line in list parameter
In { ... } list

Unexpected MODEND encountered
MODEND without preceding MODULE

Unknown option
In OPT statement

Unknown option /%c
Unknown option on Command line

Unrecognised attribute in GROUP directive

Unrecognised optimisation switch '%c'
In OPT statement or Command line

User pressed Break/Ctrl-C
Assembly aborted by user

XDEF'd symbol %n not defined
Symbol was XDEF'd but never defined

XDEF/XREF can only be used when producing linkable output

Zero length INCBIN
Warning of zero length INCBIN statement

Appendix B - Psylink Error Messages

Format: In the list below, *%x* represents the variable part of the error message, as follows:

%s is replaced by a string
%c is replaced by a single character
%d is replaced by a 16 bit decimal number
%l is replaced by a 32 bit decimal number
%h is replaced by a 16 bit hexadecimal number
%n is replaced by a symbol name
%t is replaced by a symbol type, e.g. section, symbol or group.

Linker Messages:

%t %n redefined as section
New definition of previously defined symbol

%t '%n' redefined as group
New definition of previously defined symbol

%t '%n' redefined as XDEF symbol
New definition of previously defined symbol

Attempt to switch section to %t '%n'
Non-section type symbol referenced in section switch

Attempt to use %t '%n' as a section in expression
Section type symbol required

Branch (%l bytes) is out of range
Branch instruction cannot reach target

Branch to odd address
Warning that branch instruction goes to an odd address

Code in BSS section '%n'
BSS type sections should not contain initialised data

COFF file has incorrect format
COFF format files are those produced by Sierra C cross compiler, etc.

Different processor type specified
Object code is for different processor type than target or attempt was made to link code for different processor types

Division by zero

Error closing file
DOS close file call returned error status

Error in /e option
On Command line

Error in /o option
On Command line

Error in /x option
On Command line

Error in command file

Error in Linker options
On Command line

Error in REGS expression

Error reading file %f
DOS read file call returned error status

Error writing object file
DOS write file call returned error status - probably disk full

Errors during pass 1 - pass 2 aborted
Pass 2 will not take place if there were errors on Pass 1

Expecting a decimal or hex number
/o option on Command line

File %f is in out-of-date format
File should be re-built be re-assembling

File %f is not a valid library file

File %f is not in PsyLink file format

Group '%n' is too large (%l bytes)
Group is larger than its size attribute allows

Group '%n' specified with different attributes
Different definitions of a group specify different attributes

Illegal XREF reference to %t '%n'
Object file defines xref to symbol which cannot be XREF'd, e.g. a Section name

Illegal zero length short branch
Short branches must not have offset of 0

Multiple run addresses specified
More than one run address specified

No source files specified
No source file on Command line

Object file made with out-of-date assembler
File should be re-built before re-assembling

Only built in groups can be used when making relocatable output
When /r command line option is used, only the built in groups can be used, i.e. no new group's may be defined

Option /p not available when downloading to target

Options /p and /r cannot be used together
On Command line

ORG ? can only be used when downloading output

Out of memory, Linker aborting

Previous group was not OBJ'd
Cannot specify OBJ() attribute if previous group did not have obj attribute

Reference to %t '%n' in expression
Use of, e.g. a section name in an expression

Reference to undefined symbol #%h
There is an internal error in the object file

Relocatable output cannot be ORG'd

Remainder by zero

Run-time patch to odd address
Warning that a run-time longword patch to an odd address will occur which may cause some systems to crash

SCSI card not present - linking aborted
Could not find SCSI card

SCSI drivers not loaded
PSYBIOS does not appear to be present

Section '%n' must be in one of groups code, data or BSS

Section '%n' placed in non-group symbol #%h
There is an internal error in the object file

Section '%n' placed in non-group symbol '%n'
An attempt was made to place a section in a non-group type symbol

Section '%n' placed in two different groups
Section is placed in different groups

Section '%n' placed in unknown group symbol #%h
There is an internal error in the object file

Section '%n' must be in one of groups text, data or BSS

Specified patch cannot be represented in target's relocation format
When producing relocatable code, certain run time relocations are allowed, depending on the target output file format. This error occurs when the type of patch required cannot be represented in the output file format, e.g. patching a byte in the ST file format which allows only longwords to be patched.

Symbol '%n' multiply defined
New definition of previously defined symbol

Symbol '%n' not defined
Undefined symbol

Symbol '%n' placed in non-section symbol #%h
There is an internal error in the object file

Symbol '%n' placed in unknown section symbol #%h
There is an internal error in the object file

Symbol in COFF format file has unrecognised class
COFF format files are those produced by Sierra C cross compiler, etc.

Timed out sending data to target
Target not responding or offline

Too many file names specified
Too many parameters on command line

Too many modules to link
Maximum of 256 modules may be linked

Too many symbols in COFF format file
COFF format files are those produced by Sierra C cross compiler, etc.

Unable to open output file
Could not open specified output file

Undefined symbol in COFF file patch record
COFF format files are those produced by Sierra C cross compiler, etc.

Unit number must be in range 0-127

Unknown option /%c
On Command line

Unknown processor type '%s'
Could not recognise target processor type

Unrecognised relocatable output format
/r option on command line

User pressed Break/Ctrl-C
Linking aborted by user

Value (%l) out of range in instruction patch
Value to be patched in is out of range

Appendix C - Psylib Error Messages

Librarian Messages:

Cannot add module : it already exists
Module may only appear in a library once

Could not create object file
Error creating object file when extracting

Could not create temporary file
Error creating temporary file

Could not open/create
DOS error opening file

Error reading library file
DOS error reading file

Error writing library file
DOS error writing file, probably disk full

Incorrect format in object file
Error in object file format - re-build it

No files matching
No object files matching the specifications were found

No library file specified

No object files specified

No option specified
An action option must be specified on the command line

Unknown option /
On Command line, option not recognised

Appendix D - The SPC700 Assembler

This appendix documents the **SPC700** Assembler, which is supplied with some versions of the Psy-Q Development system.

It is not an exhaustive description of the Assembler, since, functionally, it is almost identical to the ASM658 Assembler. Therefore, this section details the differences between the two programs; for any part of the program not mentioned in the following pages, refer to the ASM658 documentation.

Using the SPC700 Assembler

Assembler Command Line

For information about the list of Assembler Options that can follow the **/o** switch, see below, under the **OPT** Directive.

OPT Directive

The SPC700 Assembler supports the following options, which may follow an **OPT** statement in the source code, or a **/o** switch on the Assembler command line:

AN	Allow alternate number format.
C	Switch on Case Sensitivity
D	De-scope Local Labels on EQU, SET, etc.
E	Print lines containing errors.
L	Use **!** as leading character for Local Labels.
L*x*	Use ***x*** as leading character for Local Labels; ***x*** may not be + or -.
S	Treat equated symbols as labels
V	Write Local Labels to symbol file..
W	Print Warnings.
WS	Allow white space in operands.
X	Assume XREFs are in the section in which they are declared.

SPC700 Addressing Modes

- Absolute Addressing can be forced by prefixing the operand with an *exclamation mark* (!). For instance,

```
lda     !$23
```

- Direct Page Addressing can be forced by prefixing the operand with a *less than* (<). For instance,

```
lda     <var1
```

- If neither of the prefix operators < or ! are used, the Assembler will base the addressing mode to use on the following considerations:

- if the operand value is not known at the point that the instruction is encountered on pass 1, Absolute Addressing will be used;

- if the operand value is known, and is in the currently assumed Direct Page ($00xx or $01xx), Direct Page Addressing will be used;

- otherwise, Absolute Addressing will be used.

Index Conventions

Throughout this Index, the following applies:

- Program Names are in capitals;
- Directives and Functions are in italic capitals when the reference is to the definition;
- Subsidiary references to Directives and Functions are shown following the main entry.

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W

X

Z