OmegaSoft 6809
Relocatable Assembler
And
Linking Loader

OMEGASOFT 6809
RELOCATABLE ASSEMBLER
AND LINKING LOADER

ORDER No. MRALL1

RELEASED FOR SOFTWARE VERSION 1.3

Second printing, January 1983

CONTENTS

ASSEMBLER

The OmegaSoft 6809 Relocatable Assembler is a two pass assembler whose output is compatible with the OmegaSoft linking loader. This document serves as a reference for the OmegaSoft assembler, not as an introduction to assembly language programming.

SOURCE STATEMENT SYNTAX

Source statements consist of up to 80 characters in the following form :

LABEL OPERATION OPERAND COMMENTS

where each field is separated by one or more spaces.

LABEL FIELD

The label field must always start in column 1. If no label is present, there must be at least one space preceding the operation field. An asterisk "*" in the first column marks the line as a comment and no further processing is required on that line, otherwise, the label field defines a symbol. A valid symbol can contain at most six characters. The first character of a symbol must be one of the following: "A" thru "Z", or "." . The rest of the symbol can consist of the characters "A" thru "Z", "0" thru "9", ".", "$", or "_". The following symbols are reserved by the assembler : "A", "B", "D", "CC", "DP", "PC", "S", "U", "X", and "Y". A symbol can only be defined once in the label field. A symbol defined in this manner is assigned the value of the location counter (exceptions are the EQU directive and the RMB directive when in absolute section). Each unique label or external reference symbol requires 13 bytes in the symbol table. The maximum number of symbols is determined by the amount of memory available to the assembler and this number is reported at the end of the assembly.

OPERATION FIELD

The operation field can either be an opcode or an assembler directive. Opcodes correspond directly to the machine instructions whereas directives control the assembly process. Refer to the instruction summary for a list of opcodes.

OPERAND FIELD

The operand field must be one of the following addressing modes :

Addressing modes

1) Immediate addressing := #expression

Immediate addressing uses information that immediately follows the operation in memory. If the operation references a two byte register (D, S, U, X, Y) then a two byte immediate value is generated, otherwise a one byte value is generated. The immediate value is interpreted either as a two's complement signed value (one byte in the range -128 to 127 or two byte in the range -32768 to 32767) or as an unsigned value (one byte in the range 0 to 255 ($FF) or two byte in the range 0 to 65536 ($FFFF)).

Immediate addressing is also used to represent the specification of two registers, or a register list. The two register mode is used by the TFR and EXG instructions. The two registers are separated by a comma and must either be both one byte registers or both two byte registers. The register list mode is used by the PSH and PUL instructions. They consist of a list of registers separated by commas.

2) Relative addressing := expression

Relative addressing is used by branch instructions. The offset is a one byte value for short branches with a range of -128 to +127 bytes from the start of the next instruction. The offset is a two byte value for long branches with a range of -32768 to +32767 bytes from the start of the next instruction.

3) Extended addressing := expression

Extended addressing uses two bytes to contain the address of the operand. This allows addressing of the full memory range of $0000 to $FFFF.

4) Direct addressing := <expression

Direct addressing uses one byte to contain the lower part of the address of the operand, the upper part being contained in the direct page register. The lower address must be in the range of 0 to 255 ($FF). Notice that in this assembler you must use the "<" character to force direct addressing where extended addressing is possible.

5) Inherent addressing

Inherent addressing has no operands, all information required is contained in the operation code.

6) Indexed addressing := expression,R or [expression,R] or [expression]

Indexed addressing is relative to one of the index registers (except extended indirect). In all indexed addressing forms the value 0 can be omitted (except extended indirect).

(A) The first type of indexed addressing is constant offset from R (where R = X, Y, U, or S). Valid forms are :

expression,R and [expression,R]

where the number of extension bytes depends on the size of the expression. The "[ ]"'s indicate the indirect addressing option.

(B) Accumulator offset from R is of the following form :

acc,R or [acc,R]

where acc is A, B, or D.

(C) Auto increment/decrement R is of the following form :

0,R+ or 0,R++ or 0,-R or 0,--R or [0,R++] or [0,--R]

Notice that indirect mode can only be used with increment or decrement by two.

(D) Constant offset from program counter is of the form :

expression,PCR or [expression,PCR]

where expression can be an 8 or 16 bit offset.

(E) Extended indirect is of the form :

[expression]

where expression is a 16 bit address.

Expressions

An expression is a combination of symbols, constants, and algebraic operations. The expression is used to specify a value which is to be used as an operand. The expression is evaluated left to right and there may be no spaces in the expression. Expressions may contain absolute, relocatable, or externally defined symbols. Absolute symbols used in an expression must be defined before use. An externally defined symbol cannot be subtracted in an expression (i.e. on right side of minus sign). The result of a relocatable minus a relocatable is absolute. External and relocatable symbols can only be used with the plus and minus operations. The following operations are available :

+	addition
-	subtraction
!.	logical and
!+	logical or
!X	logical exclusive or
!<	shift left
!>	shift right

Symbols

Every symbol is assigned a 16 bit value which is used in place of the symbol during expression evaluation. The reserved symbol "*" represents the value of the location counter at the start of the line of source code. Symbols have one of the following attributes:

1) Absolute (see EQU, ASCT, and RMB directives)
2) Relocatable (in label field)
3) External reference (see XREF directive)

Constants

Four types of constants are provided in this assembler : hex, integer, binary, and character. Hex values are preceded by a "$" and can be in the range of $0 thru $FFFF. Integer values have no base specifier (prefix) and can be in the range of -32767 thru +32767. Binary values are preceded by a "%" and can be in the range of %0 thru %1111111111111111. A fourth type of non-numeric data, character, is also provided. Character data is represented by a single character preceded by a "'". The result is a value which is the ASCII value of the character.

SPECIAL OPCODES

The following opcode mnemonics are automatically translated into their corresponding 6809 equivalent.

Mnemonic	6809 Equivalent	Meaning
CLC	ANDCC #$FE	clear carry flag
SEC	ORCC #$01	set carry flag
SCALL expr	SWI[x]	
	FCB expr	system call, example :
		OS9 only : SCALL $4D =
		SWI2
		FCB $4D
		others : SCALL $4D =
		SWI
		FCB $4D

ASSEMBLER DIRECTIVES

ASCT - absolute section

syntax : ASCT [comment]

The ASCT directive will instruct the assembler that any further RMB directives are to be in absolute section - non relocatable. This is useful for assigning addresses to I/O devices or variables. The ORG directive can only be used when in absolute section.

BSZ - block storage of zeroes

syntax : [label] BSZ expression [comment]

The BSZ directive causes the assembler to allocate a block of bytes, each with an initial value of zero. The value of the expression must be absolute and the expression value must be defined before the BSZ directive statement.

END - end of source program

syntax : END [label]

The END directive indicates the logical end of the source program has been encountered. If the operand is specified the relative address of the label is used as the begin execution address of this program (on OS9 this label is ignored).

ENDC - end of conditional assembly

syntax : ENDC [comment]

This directive marks the end of conditional assembly, the assembly is now enabled.

EQU - equate symbol to a value

syntax : label EQU expression [comment]

The EQU directive assigns the value of the expression in the operand field to the label. The expression cannot contain any external references, or forward references. The symbol will be absolute (non relocatable).

FCB - form constant byte

syntax : [label] FCB expression{,expression} [comment]

The value of each expression is truncated to one byte and stored in successive locations in the object program. The expression(s) must be absolute.

FCC - form constant characters

syntax : [label] FCC number,string
or
[label] FCC <delimiter>string<delimiter>

The FCC directive stores ASCII characters into consecutive bytes of memory. In the first format the number defines the number of characters after "," to be output. There are never more characters output than there are characters on a line regardless of the count specified.

The second format of the FCC directive specifies the characters to be output between 2 identical delimiters. The delimiter is the first non-blank character after the "FCC".

FDB - from double byte

syntax : [label] FDB expression{,expression} [comment]

The value of each expression fills 2 bytes and are stored in successive locations in the output. The expression(s) may be of type absolute, relocatable, or external.

IFxx - conditional assembly

syntax : IFxx expression [comment]

This set of directives is used to determine if the code that follows is to be assembled. This is effective until a "ENDC" directive is encountered. These directives cannot be nested. There are six variations of the IFxx directive :

IFEQ - assemble code if expression is equal to 0
IFGE - assemble code if expression is greater than or equal to 0
IFGT - assemble code if expression is greater than 0
IFLE - assemble code if expression is less than or equal to 0
IFLT - assemble code if expression is less than 0
IFNE - assemble code if expression is not equal to 0

INCL - include source from file

syntax : INCL <path name>

The path name given is opened and source is read from the path until end of file. At that point the path is closed and source continues from the main file. Include directives may not be nested.

NAM - assign program name

syntax : NAM string [comment]

The NAM directive specifies the name of the relocatable program module. Only the first 6 characters of the specified string are used. If no directive is specified, a default blank name is output.

OPT - assembler options

syntax : OPT options{,option}

The OPT directive is used to control the format of the assembler listing output. Options not recognized by this assembler are ignored. The following is a list of options recognized :

L print the listing from this point on (default). This only has an effect if the "L" command line option is used.

NOL do not print the listing from this point on.

W Do issue warning messages (default). This only has an effect if the "W" command line option is used.

NOW Do not issue warning messages.

ORG - set location counter

syntax : ORG expression [comment]

This directive is only valid when in absolute section (ASCT). The absolute section location counter will be set to the value of the expression. The expression must be absolute.

PAGE - move listing to next page

syntax : PAGE

This directive causes the listing to move to the top of the next page if the listing is enabled and page size is non-zero.

PSCT - program section

syntax : PSCT

This directive will change the section to program section (relocatable). The default at the start of the assembly is program section.

RMB - reserve memory bytes

syntax : [label] RMB expression [comment]

The operation of this directive depends on which section the assembler is in. If the assembler is in program section (PSCT) then this directive is identical in operation to BSZ. If the assembler is in absolute section (ASCT) then the absolute section location counter will be used as the absolute value of the symbol and the value of the expression will be added to the absolute section location counter.

TTL - set page title

syntax : TTL string

The TTL directive causes the page title to be set to the string in the operand field.

XDEF - external definition

syntax : XDEF symbol{,symbol} [comment]

The XDEF directive is used to specify that the list of symbols is defined within the current program and the definition is passed through the linker.

XREF - external reference

syntax : XREF symbol{,symbol} [comment]

The XREF directive is used to specify that the list of symbols is referenced within the current program but is defined in another program (via XDEF).

INSTRUCTION SUMMARY

INSTR	DESCRIPTION	IMM	DIR	EXT	IDX	INH	REL
ABX	B (UNSIGN) + X -> X					X	
ADCA	A + M + c -> A	X	X	X	X		
ADCB	B + M + c -> B	X	X	X	X		
ADDA	A + M -> A	X	X	X	X		
ADDB	B + M -> B	X	X	X	X		
ADDD	D + M:M+1 -> D	X	X	X	X		
ANDA	A and M -> A	X	X	X	X		
ANDB	B and M -> B	X	X	X	X		
ANDCC	CC and M -> CC	X					
ASLA	A } <---					X	
ASLB	B } [] <- [] [] [] <- 0					X	
ASL	M } c b7 b0		X	X	X		
ASRA	A } --->					X	
ASRB	B } [] [] [] -> []					X	
ASR	M } b7 b0 c		X	X	X		
BCC	branch if carry clear						X
BCS	branch if carry set						X
BEQ	branch if equal to zero						X
BGE	branch if >= zero (signed)						X
BGT	branch if > zero (signed)						X
BHI	branch if > zero (unsigned)						X
BHS	branch if >= zero (unsigned)						X
BITA	M and A sets CC	X	X	X	X		
BITB	M and B sets CC	X	X	X	X		
BLE	branch if < zero (signed)						X
BLO	branch if <= zero (unsigned)						X
BLS	branch if <= zero (signed)						X
BLT	branch if < zero (signed)						X
BMI	branch if minus (b7 set)						X
BNE	branch if not equal to zero						X
BPL	branch if plus (b7 clear)						X
BRA	branch always						X
BRN	branch never						X
BSR	branch to subroutine						X
BVC	branch if overflow clear						X
BVS	branch if overflow set						X

INSTR	DESCRIPTION	IMM	DIR	EXT	IDX	INH	REL
CLC	0 -> c					X	
CLRA	0 -> A					X	
CLRB	0 -> B					X	
CLR	0 -> M		X	X	X		
CMPA	A - M sets CC	X	X	X	X		
CMPB	B - M sets CC	X	X	X	X		
CMPD	D - M:M+1 sets CC	X	X	X	X		
CMPS	S - M:M+1 sets CC	X	X	X	X		
CMPU	U - M:M+1 sets CC	X	X	X	X		
CMPX	X - M:M+1 sets CC	X	X	X	X		
CMPY	Y - M:M+1 sets CC	X	X	X	X		
COMA	not A -> A					X	
COMB	not B -> B					X	
COM	not M -> M		X	X	X		
CWAI	CC and M then wait	X					
DAA	Decimal adjust A					X	
DECA	A - 1 -> A					X	
DECB	B - 1 -> B					X	
DEC	M - 1 -> M		X	X	X		
EORA	A eor M -> A	X	X	X	X		
EORB	B eor M -> B	X	X	X	X		
EXG	reg1 <--> reg2	X					
INCA	A + 1 -> A					X	
INCB	B + 1 -> B					X	
INC	M + 1 -> M		X	X	X		
JMP	addr -> PC		X	X	X		
JSR	JUMP to subroutine		X	X	X		
LBCC	branch if carry clear						X
LBCS	branch if carry set						X
LBEQ	branch if equal to zero						X
LBGE	branch if >= zero (signed)						X
LBGT	branch if > zero (signed)						X
LBHI	branch if > zero (unsigned)						X
LBHS	branch if >= zero (unsigned)						X
LBLE	branch if < zero (signed)						X
LBLO	branch if <= zero (unsigned)						X

INSTR	DESCRIPTION	IMM	DIR	EXT	IDX	INH	REL
LBLS	branch if <= zero (unsigned)						X
LBLT	branch if < zero (signed)						X
LBMI	branch if minus (b7 set)						X
LBNE	branch if not equal to zero						X
LBPL	branch if plus (b7 clear)						X
LBRA	branch always						X
LBRN	branch never						X
LBSR	branch to subroutine						X
LBVC	branch if overflow clear						X
LBVS	branch if overflow set						X
LDA	M -> A	X	X	X	X		
LDB	M -> B	X	X	X	X		
LDD	M:M+1 -> D	X	X	X	X		
LDS	M:M+1 -> S	X	X	X	X		
LDU	M:M+1 -> U	X	X	X	X		
LDX	M:M+1 -> X	X	X	X	X		
LDY	M:M+1 -> Y	X	X	X	X		
LEAS	addr -> S				X		
LEAU	addr -> U				X		
LEAX	addr -> X				X		
LEAY	addr -> Y				X		
LSLA	A } <---					X	
LSLB	B } [] <- [] [] [] <- 0					X	
LSL	M } c b7 b0		X	X	X		
LSRA	A } --->					X	
LSRB	B } 0 -> [] [] [] -> []					X	
LSR	M } b7 b0 c		X	X	X		
MUL	A * B -> D					X	
NEGA	not A + 1 -> A					X	
NEGB	not B + 1 -> B					X	
NEG	not M + 1 -> M		X	X	X		
NOP	no operation					X	
ORA	A or M -> A	X	X	X	X		
ORB	B or M -> B	X	X	X	X		
ORCC	CC or M -> CC	X					
PSHS	push reg. list on S stack	X					
PSHU	push reg. list on U stack	X					
PULS	pull reg. list off S stack	X					
PULU	pull reg. list off U stack	X					

INSTR	DESCRIPTION	IMM	DIR	EXT	IDX	INH	REL
ROLA	A } ----------					X	
ROLB	B } -[] <- [] [] [] <---					X	
ROL	M } c b7 b0		X	X	X		
RORA	A } ----------					X	
RORB	B } ---> [] -> [] [] []-					X	
ROR	M } c b7 b0						
RTI	return from interrupt					X	
RTS	return from subroutine					X	
SBCA	A - M - c -> A	X	X	X	X		
SBCB	B - M - c -> B	X	X	X	X		
SCALL	system call M	X					
SEC	1 -> c					X	
SEX	sign extend B into A					X	
STA	A -> M		X	X	X		
STB	B -> M		X	X	X		
STD	D -> M:M+1		X	X	X		
STS	S -> M:M+1		X	X	X		
STU	U -> M:M+1		X	X	X		
STX	X -> M:M+1		X	X	X		
STY	Y -> M:M+1		X	X	X		
SUBA	A - M -> A	X	X	X	X		
SUBB	B - M -> B	X	X	X	X		
SUBD	D - M:M+1 -> D	X	X	X	X		
SWI	software interrupt 1					X	
SWI2	software interrupt 2					X	
SWI3	software interrupt 3					X	
SYNC	synchronize to interrupt					X	
TFR	reg1 -> reg2	X					
TSTA	test A sets CC					X	
TSTB	test B sets CC					X	
TST	test M sets CC		X	X	X		

DEFINITIONS :

A, B, D, X, Y, U, S, CC, PC ____ contents of 6809 registers
c _____ carry bit in CC
M _____ 8 bit contents of memory
M:M+1 _____ 16 bit contents of memory

ERROR MESSAGES

174 Invalid auto increment/decrement format.
Single auto increment or decrement was specified in the indirect mode (e.g., LDB [Y+]).

175 Invalid index register format.
One of the accumulators was specified as the offset in the index mode but was not followed by one of the index registers.

176 Invalid expression for PSH/PUL.
The register list following one of the instructions PSHS, PULS, PSHU, or PULU contained symbols that were not registers.

177 Incompatible register for PSH/PUL instruction.
The register list for the PSHS/PULS instructions contained the register "S" or the register list for the PSHU/PULU instructions contained the register "U".

178 Invalid register operand specification.
Undefined register name encountered in indexed addressing mode.

179 Incompatible register pair.
The register pair of an EXG or TFR instruction was not the same size. The pair must be two 16 bit registers or two 8 bit registers.

202 Label or opcode error.
A label or opcode symbol does not begin with an alphabetic character or period.

203 Error in operand expression.
Incompatible symbol types in expression.

204 Operand needed.
An operand was not found when expected.

205 Label error.
Invalid character in label.

207 Undefined opcode.
The symbol in the opcode field is not a valid opcode or directive.

208 Branch out of range.
The operand resulted in an offset greater than 129 bytes forward or 126 bytes backward from the first byte of the branch instruction.

209 Illegal addressing mode.
The specified addressing mode in the operand field is not valid with this instruction type.

210 Byte overflow.
The operand's value exceeded one byte. The most significant 8 bits of the 16 bit expression must all be zeroes or all ones for a one byte two's complement field.

212 Directive operand error.
A syntax error was detected in the operand field of a directive.

214 FCB directive syntax error.
The structure of the FCB directive is syntactically incorrect.

215 FDB directive error.
The structure of the FDB directive is syntactically incorrect.

216 Directive operand error.
The directive's operand field is missing, terminated by an invalid terminator, or an expression in the operand field contains an invalid operator.

219 No END statement.
The END directive was not found at the end of the source file. The END directive is automatically supplied.

222 Symbol table overflow.
The symbol table has overflowed. This is a fatal error and terminates the assembler during pass one.

223 The directive must or must not have a label
Depending on the directive used, the label field must be blank or must contain a valid symbol.

224 Multiply defined symbol.
An attempt was made to define a symbol that was already defined.

241 Illegal symbol used in an expression.
An undefined, forward referenced, external reference, or relocatable symbol was used illegally in an expression that does not allow them.

243 XREF or XDEF directive operand error.
An invalid symbol or no operand was detected in the operand field of the XDEF or XREF directive.

WARNING MESSAGES

1 Long branch not required.
The destination could have been reached with a short branch. This warning can also be issued when using PCR relative addressing.

7 Extended addressing.
This could result in non position independent code.

8 Non absolute immediate addressing.
This could result in non position independent code.

OBJECT MODULE FORMAT

The following is a description of the relocatable object module format used by this assembler and the companion linking loader. The file is recorded in a binary record format where a record consists of :

D L X X X .... X X X C CR

where :
D is the ASCII character "D" and signifies start of record.
L is a byte that is the length (data plus checksum).
X is the data of the record.
C is the 2's complement checksum (starting with L).
CR is a carriage return ($D).

Record types :

2) Header

'2' : $00 : NAME : "OB"

This record is the first record of the object module. The six character module name (NAME) is the name that was specified in the NAM directive.

3) External symbol definition (ESD)

'3' : SYMTYPE/SECT : DEF

where SYMTYPE is the upper nibble and has the following format:
$0 - load section definition
$2 - label definition
$3 - external reference

SECT is the lower nibble and has the following format :
$0 - ASCT or any (for XREF)
$1 - BSCT - not implemented - size set to 0
$2 - CSCT - not implemented - size set to 0
$3 - DSCT - not implemented - size set to 0
$4 - PSCT

This record marks definitions of XREFs, XDEFs, and the size of the code. If SYMTYPE = 0 and SECT = 0 then DEF is a 2 byte ASCT section length (always zero) and a 2 byte ASCT start location (always zero). If SYMTYPE = 0 and SECT = 1 then DEF is a 2 byte BSCT section length (always zero). If SYMTYPE = 0 and SECT = 2 then DEF is a 2 byte CSCT section length (always zero). If SYMTYPE = 0 and SECT = 3 then DEF is a 2 byte DSCT section length (always zero). If SYMTYPE = 0 and SECT = 4 then DEF is a 2 byte value of the total PSCT size. If SYMTYPE = 2 (XDEF) then DEF is a 6 byte name followed by the 2 byte relative address (if SECT = 4 PSCT) or its absolute address (if SECT = 0 ASCT). If SYMTYPE = 3 (XREF) then DEF is a 6 byte name and SECT = 0 (any section will match). SYMTYPE/SECT : DEF can be repeated 0 or more times.

4) Program

'4' : ESD INDEX : $00 : RELADDR : BYTES

This record contains program code. ESD INDEX will always be $0004 which indicates the code is in PSCT. RELADDR is the 2 byte address where the data starts to load relative to the start of the module. BYTES are up to 122 bytes of code to load.

5) Relocation and Linking record

'5' : FLG : RELADDR : ESD INDEX

This record marks addresses which must be relocated by the linker and marks addresses which have external references to resolve. This record corresponds to program bytes which were included in the last type '4' record. FLG is a 1 byte flag that is $00 if the relocation is to be added, or $08 if the relocation is to be subtracted. RELADDR is the relative address of the module word that is to be relocated. ESD INDEX indicates which external symbol definition (ESD) corresponds with this relocation. 0 means the 1st ESD physically in the module, 1 the 2nd ESD, etc.. FLG : RELADDR : ESD INDEX can be repeated zero or more times.

6) Terminator

'6' : SECT : RELADR

This record marks the end of the object module. RELADR is the start execution offset referenced to the start of section SECT.

LINKER

The OmegaSoft Linking Loader is a two pass loader which can accept relocatable object modules from the OmegaSoft Relocatable Assembler. On the first pass all external symbol values are defined (and relocated if necessary), all object modules which are to be loaded from libraries are determined, and the size of the load module is determined. On pass two the load module is produced using the information from pass 1, satisfying external references, and relocating specified addresses. Up to 200 modules may be handled by this linker. The number of global symbols that can be handled is determined by the amount of memory available to the linker.

COMMANDS

Each command cannot exceed 80 characters. Pass one terminates when the OBJA command is entered at which point all commands entered during pass one will be repeated and echoed by the linker until the OBJA command is encountered the second time. At this point entry resumes from the input path for acceptance of the map command (if used) and the exit command. No more than 20 commands may be used between the first command and the OBJA command, inclusive.

CURP - set current location

syntax : CURP=[\]$<hex number>

This command modifies the current loader location counter. If the '\' is not specified then the location counter will be set to <hex number> + start load address. If the '\' is specified then future modules will be loaded at an even power of two relative to the start of the section. For example : CURP=\$100 will cause future modules to load at addresses xx00 and CURP=\$10 will cause future modules to load at addresses xxx0. This option remains in effect until another CURP command is encountered.

DEF - loader symbol definition

syntax : DEF:<name>=$<hex number>

This command defines a global symbol and inserts it into the global symbol table. The symbol is defined as absolute (ASCT).

ENDP - ending PSCT address

syntax : ENDP=$<hex number>

This command sets the end load address of the load module. If the actual module would be smaller, then the load module is padded with zeroes (memory image operating systems only). The default for ENDP is $FFFF.

ERRORS : If the module exceeds the ENDP address, then a 'ENDP ADDRESS EXCEEDED' error message will result.

EXIT - exit linker

syntax : EXIT

This command terminates the linker.

LIB - library search

syntax : LIB=<file name1>{,<file nameN>}

This command directs the loader to load object modules in the specified files only if the object module satisfies an unresolved external reference from an object module previously loaded. The file names may be separated by spaces or commas.

ERRORS : A 'MULTIPLY DEFINED SYMBOL' error can result if the same name is defined in more than one loaded module. An 'UNDEFINED SYMBOL' error can result if there is no name definition to satisfy a reference.

LOAD - load file

syntax : LOAD=<file name1>{,<file nameN>}

This command directs the linker to load the specified files. There can be one or more object modules in a file.

ERRORS : A 'MULTIPLY DEFINED SYMBOL' error can result if the same name is defined in more than one loaded module. An 'UNDEFINED SYMBOL' error can result if there is no name definition to satisfy a reference.

MAP - print load map

syntax : MAP[U][S][M][D]
MAPC
MAPF

This command displays the current state of the modules loaded. The U option will list any undefined symbols. The S option will list the memory size of the modules loaded plus symbol table usage. The M option will list the starting load address of all modules loaded. The D option will list each loaded module along with the external definitions contained in that module. MAPC is equivalent to MAPS. MAPF is equivalent to MAPUSMD.

MO - map output

syntax : MO=<path>

This command overrides the load map specification in the command line. Any path valid in the command line redirection is valid with this command.

OBJA - produce load module

syntax : OBJA=<file name>

This command specifies the output load module produced by the linker. The OBJA command terminates pass one of the linker, and consequently the start of pass two. The output file is filled during pass two and is closed upon executing the OBJA command during pass two of the linker.

STRP - starting PSCT address

syntax : STRP=$<hex number>

This command sets the starting load address of the load module. The default is $0.

EXAMPLE

In the OmegaSoft Pascal system there are normally three main components in the resultant pascal program : stack setup, main program, and runtime library. This example will use the hypothetical program 'test'.

In the setup code is some code like this :

```
 XDEF START
 XREF TEST
    |
    |
 LBSR TEST
```

In the main program :

```
 XDEF TEST
TEST EQU *
    |
    |
 LBSR RUN1
    |
 LBSR RUN2
    |
 XREF RUN1,RUN2
```

where RUN1 and RUN2 are routines in the runtime library.

In the runtime library :

```
 XDEF RUN1
RUN1 EQU *
   |
   |
 XDEF RUN2
RUN2 EQU *
   |
   |
```

By assembling the stack setup code (.PS), the output of the compiler for the main program (.CA), and merging the assembled output of all the runtime libraries together (RL), these can be used as input to the linker as in :

```
LOAD=TEST.PS TEST.CA
LIB=RL
OBJA=TEST
```

The linker will build a loadable version of the pascal program with only the necessary runtime routines and with all of the LBSR XXXX instructions replaced with the correct relative offset into the definitions in the other modules.