COFF Programming Utilities Guide

for FLEXOS™ 386

Beta Edition: June 1987

Software Version: FlexOS 386 1.0

nnnn-nnnn-001

COPYRIGHT

DISCLAIMER

DIGITAL RESEARCH MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Digital Research Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research Inc. to notify any person of such revision or changes.

NOTICE TO USER

From time to time changes are made in the filenames and in the files actually included on the distribution disk. This manual should not be construed as a representation or warranty that such files or materials and facilities exist on the distribution disk or as part of the materials and programs distributed. Most distribution disks include a "README.DOC" file. This file explains variations from the manual which do constitute modification of the manual and the items included therewith. Be sure to read this file before using the software.

TRADEMARKS

Digital Research and its logo are registered trademarks of Digital Research Inc. FlexOS is a trademark of Digital Research Inc. We Make Computers Work is a service mark of Digital Research Inc. CASM, CLINK, CLIB, and CSID are trademarks of Digital Research. Intel is a registered trademark of Intel Corporation.

Foreword

The COFF Programming Utilities Guide for the FlexOS(TM) 386 (hereinafter cited as the COFF Utilities Guide) assumes that you are familiar with the FlexOS operating system environment. It also assumes that you are familiar with Intel® 80386 processor architecture and assembly language programming as described in the Intel 80386 Programmer's Reference Manual, (Order Number 230985-001).

The COFF Utilities Guide describes several programs that aid programmers and system designers in developing software.

Chapter 1 describes CASM™, an assembler that translates 80386 assembly language statements into a relocatable object file. Chapter 2 describes the essential elements of CASM assembly language. Chapter 3 describes CASM directives that control code generation, linkage, conditional assembly, etc.

Chapter 4 describes CLINK™, a linkage editor that combines object modules into an executable file that can be loaded by FlexOS.

Chapter 5 describes CLIB™, a librarian that can create and manage libraries of program modules.

Chapter 6 describes CSID™, a symbolic debugger that can load and execute programs, display memory, set breakpoints, etc. Chapter 7 describes how CSID handles expressions. Chapters 8 and 9 describe the various CSID commands used to debug code.

Appendix A describes COFF, the Common Object File Format. Appendix B contains a listing of a sample CASM program.

Contents

Tables

Figures

CASM - COFF ASSEMBLER

CASM™ converts a source file containing 80386 assembly language instructions into a machine language object file in COFF[1] format. In addition to the object file, CASM can produce a list file containing the assembly language listing with any error messages.

CASM produces the output files using the same filename as the source file. For example, if the name of the source file is DRIVER.A, CASM produces the files DRIVER.O, and DRIVER.LST.

1.1 CASM COMMAND LINE

Invoke CASM with the following command form:

```
CASM filespec [-Ostring] [-Ostring] ...
```

where **filespec** is the name of the source file to be compiled, which can include an optional path specification denoting the file's location. A path specification is not needed if the source is in the current directory. If you do not specify a filetype, CASM assumes filetype .A. **-Ostring** is an option string that controls CASM operation as described below.

1.2 COMMAND-LINE OPTIONS

Table 1-1 contains a summary of the CASM command-line options, described in detail in the following subsections. Options can be entered in any order. Options that do not require a parameter may be grouped following a single hyphen with no space between letters. Options that require a parameter must be entered separately with no space between the option letter and its parameter.

If you specify an invalid option in the command line, CASM displays:

```
Invalid command line option
```

Table 1-2 lists the error messages output by CASM.

[1] COFF is an acronym for Common Object File Format; see Appendix A for a detailed explanation of COFF.

Table 1-1. CASM Command-line Options

Parameter	Explanation
B	Output absolute binary data (no COFF)
C	Convert symbols to uppercase
Dsymbolname	Define symbol in command line
Ipath\filename	Include file specified by **pathname** into assembly at beginning of module
J	Suppress JMP optimization
L	Output local symbols
N[n]	Include line numbers on every nth line number starting with the first line in the object file
Opath\filename	Set path\filename for object (.O) file
P[path\filename]	Set path\filename for list (.LST) file
V	Display assembler banner and version

1.2.1 B Parameter

The B parameter directs CASM to output only binary machine code without any COFF information. If the source code does not reference any external labels or variables and needs no relocation, the object code output by CASM should be executable, although not directly loadable by the FlexOS 386 program loader.

1.2.2 C Parameter

The C parameter directs CASM to convert all symbols to uppercase.

1.2.3 Dsymbolname Parameter

The D parameter directs CASM to use the symbol defined by **symbolname** in the command line as if it had been defined EXTRN in the source code.

1.2.4 Ipath\filename Parameter

The I parameter directs CASM to include the contents of a specified file at the beginning of the module being assembled. The file must be identifed by a valid filename preceded by an I (upper case i). If no filename extension is specified, CASM assumes an extension of .A.

If you do not specify a path, CASM searches for the file in the current directory; if unsuccessful, it searches the directory containing the source file.

1.2.5 J Parameter

The J parameter directs CASM to not perform any code optimization for JMP instructions.

1.2.6 L Parameter

The L parameter directs CASM to include local symbols in the object file.

1.2.7 N[n] Parameter

The N[n] parameter directs CASM to include line number debugging symbols in the object file on every **nth** line beginning with line one. The default is n = 5.

1.2.8 Opath\filename Parameter

The O parameter directs CASM to output the object file to the specified path\filename.

1.2.9 P[path\filename] Parameter

The P parameter directs CASM to output a list file. If you do not specify **path\filename**, the -P option outputs the list file using source's path\filename with filetype LST.

1.2.10 V Parameter

The V parameter directs CASM to display the logon banner with version number.

1.3 EXAMPLE COMMANDS

The following are example CASM command lines:

A>casm test.a -jb
Assemble the source file "test.a", suppress JUMP optimization and output only binary code to the object file.

A>casm test -n -pb:\caslist
Assemble the source file "test.a", and send the listing file "test.lst" containing line numbers every fifth line to the directory "caslist" on drive B.

A>casm test -Idefines
Assemble the source file "test.a", and include the file "defines.a".

A>casm test -cl Assemble the source file "test.a", convert all the symbols to uppercase, and include local symbols in the object file.

A>casm test -dmy_variable

Assemble the source file "test.a", and define the symbol "my_variable" as EXTRN. This is equivalent to using the statement

```
extrn   my_variable
```

in the source code.

1.4 CASM ERROR MESSAGES

Table 1-2. CASM Error Messages

Error Message	Cause
Syntax error	General purpose error message issued whenever CASM cannot properely parse an instruction or mnemonic. Check the syntax.
Ambiguous operand	The size of an instruction is not specified (BYTE, WORD, or LONG). Use the "type" operator.
Missing closing quote around string	There is no closing delimiter around a string. Supply the delimiter.
Symbol doubly defined	All variables and labels must have unique names. Use another name.
Initial value out of range	A value supplied with a DB, DW, or DL directive is out of the range of values the variable can contain. Change the value or use different directive.
Invalid expression	CASM detected an error when parsing the expression. Check the syntax.
Invalid directive: ORG	CASM detected an error when parsing the argument to an ORG directive. Check the syntax.

Table 1-2. (Continued)

Error Message	Cause
Invalid directive: EQU	CASM detected an error when parsing the argument to an EQU directive. Check the syntax.
Bad type	An invalid type was supplied to an EXTRN directive. Check the type.
ELSE directive with no corresponding IF	Self explanatory. Supply the missing IF clause.
ENDIF directive with no corresponding IF	Self explanatory. Supply the missing IF clause.
Invalid directive: TITLE	CASM detected an error when parsing the argument to a TITLE directive. Check the syntax.
Invalid directive: ALIGN	CASM detected an error when parsing the argument to an ALIGN directive. Check the syntax.
Invalid directive: LINE	CASM detected an error when parsing the argument to a LINE directive. Check the syntax.
Invalid directive: SECTION	CASM detected an error when parsing the argument to a SECTION directive. Check the syntax.
More than the maximum number of sections declared in this file	CASM supports a maximum of 9 sections.
Symbol defined by the EQU directive used before its declaration	Self explanatory. Change source text.
Symbol missing from expression	A symbol name is missing from a directive whose argument requires such a name. Supply the missing symbol name.

Table 1-2. (Continued)

Error Message	Cause
Illegal operand in expression	The argument to an isolation operator is not the correct size. Check the argument.
Missing operand in expression	An expression or directive does not contain a necessary argument. Supply the argument.
Register used illegally in expression	The name of a CPU register was used in an expression. Check the syntax.
Closing parenthesis missing from expression	Self explanatory. Supply the missing delimiter.
Types are mis-matched in instruction	This error can occur in several contexts. For example, the contents of a WORD register were stored in a BYTE variable. Check the instruction.
No instruction on line	CASM did not detect an instruction mnemonic, directive, or comment on the line being processed. Check the source text.
Bad DD directive	The argument to the directive is not in the form: constant:constant. Check the sytax.
Bad DP directive	The argument to the directive is not in the form: constant:constant. Check the sytax.

End of Section 1

ELEMENTS OF CASM ASSEMBLY LANGUAGE

This section describes the following elements of CASM assembly language:

- character set
- tokens and separators
- delimiters
- constants
- identifiers
- operators
- expressions
- statements

Also included is a discussion of memory addressing modes, instruction prefixes, and jump instruction optimizations.

2.1 CHARACTER SET

Table 2-1 lists the CASM character set.

Table 2-1. CASM Character Set

Alphanumeric Characters	Special Characters
uppercase A - Z lowercase a - z numerals 0123456789	+ - * / = ( ) [ ] ; ' . ! , _ : $?
Nonprinting Characters	
space, tab, carriage return, and line-feed	

Only alphanumerics, special characters, and spaces can appear in a string.

Note: CASM treats lowercase letters as uppercase, except within strings and symbols. You can use the -C option to convert symbols to uppercase.

2.2 TOKENS AND SEPARATORS

A **token** is the smallest meaningful unit of a source program. Examples of tokens are instruction mnemonics, operators, symbol and register names. Adjacent tokens within the source are commonly separated by a blank character or space. Any sequence of spaces can appear wherever a single space is allowed.

CASM recognizes horizontal tabs as separators and interprets them as spaces. CASM expands tabs to eight spaces in the listing file.

2.3 DELIMITERS

Delimiters mark the end of a token and add special meaning to the instruction; separators merely mark the end of a token. When a delimiter is present, separators need not be used. However, using separators after delimiters can make source code easier to read.

Table 2-2 describes CASM separators and delimiters. Some delimiters are also operators. Operators are described in Section 2.8.

Table 2-2. Separators and Delimiters

Character	Name	Use
;	semicolon	starts comment field
:	colon	identifies a label; used in segmented address constant specification
.	period	forms variables from numbers
$	dollar sign	notation for present value of location counter; legal, but ignored in identifiers or numbers
+	plus	arithmetic operator for addition
-	minus	arithmetic operator for subtraction
*	asterisk	arithmetic operator for multiplication
/	slash	arithmetic operator for division
@	at	legal in identifiers
_	underscore	legal in identifiers
!	exclamation point	logically terminates a statement, allowing multiple statements on a single source line
'	apostrophe	delimits string constants

Table 2-2. (continued)

Character	Name	Use
20H	space	separator
09H	tab	legal in source files, expanded in list files
CR	carriage return	terminates source lines
LF	line-feed	legal after CR; if in source lines, it is interpreted as a space

2.4 DATA

Data can be either constants or variables and can be expressed in a variety of storage formats. The storage format determines how the data is internally represented and used by the processor. CASM performs type-checking to insure instructions match declared operand types. You can override type-checking by using the "type" operator (see Section 2.8.8).

Table 2-3 describes the data types supported by CASM.

Table 2-3. CASM Data Types

Type	Storage Size (bytes)	Range
BYTE	8	-128 to 127
WORD	16	-32768 to 32767
LONG	32	-2^{31} to $2^{31}-1$
DWORD	32	0 to $2^{16}-1$: 0 to $2^{16}-1$ 16-bit segment : 16-bit offset
PWORD	48	0 to $2^{16}-1$: 0 to $2^{32}-1$ 16-bit segment : 32-bit offset
QWORD	64	Numeric Data Processor (NDP) long real
TWORD	80	Numeric Data Processor (NDP) temporary real

2.5 CONSTANTS

A constant is a value known at assembly time that does not change when the program runs. A constant can be either a numeric value or a character string.

2.5.1 Numeric Constants

A numeric constant is a 32-bit integer value expressed in one of several bases. The base, called the **radix** of the constant, is denoted by a trailing radix indicator. Radix indicators can be uppercase or lowercase. CASM assumes that any numeric constant not terminating with a radix indicator is a decimal constant. Table 2-4 shows the radix indicators.

Table 2-4. Radix Indicators for Constants

Indicator	Constant Type	Base
B	binary	2
O	octal	8
Q	octal	8
D	decimal	10
H	hexadecimal	16

Binary constants must be composed of zeros and ones. Octal digits range from 0 to 7; decimal digits range from 0 to 9. Hexadecimal constants contain decimal digits and the hexadecimal digits A (10_D), B (11_D), C (12_D), D (13_D), E (14_D), and F (15_D). The leading character of a hexadecimal constant must be a decimal digit so CASM doesn't confuse a hex constant with an identifier. The following are valid numeric constants:

```
1234     1234D    1100B    1111000011110000B
1234H    0FFEH    3377O    13772Q
33770    0FE3H    1234d    0ffffh
```

2.5.2 Decimal Real

A decimal real constant is a fraction, which may be followed by an exponent. If no exponent is supplied, a decimal point is required. The exponent starts with E. The following are examples of valid decimal real constants:

```
1.4414
.001
1.0E23
9.
2E-3
```

2.5.3 Character String Constants

A character string constant is a string of ASCII characters delimited by apostrophes. All CASM instructions allowing numeric constants as arguments accept only one-, two-, or four-character constants as valid arguments. All instructions treat a one-character string as an 8-bit number, a two-character string as a 16-bit number, and a four-character string as a 32-bit number. In multi-byte strings, the value of the first character is in the high-order byte, and the value of the last character is in the low-order byte.

The numeric value of a character is its ASCII code. CASM does not translate case in character strings, so you can use both uppercase and lowercase letters. Note that CASM allows only alphanumerics, special characters, and spaces in character strings.

A DB directive is the only CASM statement that can contain strings longer than four characters (see Section 3.7.1). The string cannot exceed 255 bytes. If you want to include an apostrophe in the string, you must enter it twice. CASM interprets two apostrophes together as a single apostrophe. Table 2-3 shows valid character strings and how they appear after processing.

Table 2-5. String Constant Examples

String in source text	As processed by CASM
'a'	a
'Ab''Cd'	Ab'Cd
''''	'
'ONLY UPPER CASE'	ONLY UPPER CASE
'only lower case'	only lower case

2.6 SEGMENTED ADDRESS CONSTANTS

CASM supports segmented address constants of the form:

numeric constant : numeric constant

The colon signifies a segmented constant. For example, given the definition:

```
CALL_GATE    equ    0067:1234BAC0h
```

the number can be stored only in a data type of PWORD (16:32).

```
target    dp    CALL_GATE
```

Such a constant can also be used in immediate control transfers. For example,

```
callf      CALL_GATE
```

Likewise, a constant such as:

```
ADD1        equ  1000:55AAh
```

can be stored in a DWORD (16:16) or PWORD (16:32) as follows:

```
info1       dd   ADD1
info2       dp   ADD1
```

2.7 IDENTIFIERS

The following rules apply to all identifiers:

- Identifiers can be up to 80 characters long.
- The first character must be alphabetic or one of these special characters: ?, @, or _ .
- Any subsequent characters can be either alphabetic, numeric, or one of these special characters: ?, @, _, or $. CASM ignores the special character $ in identifiers, so that you can use it to improve readability in long identifiers. For example, CASM treats the identifier `interrupt$flag` as `interruptflag`.

There are two types of identifiers:

- Keywords
- Symbols

Keywords have predefined meanings to CASM. Symbols are identifiers you define yourself.

2.7.1 Keyword Identifiers

Keywords are reserved for use by CASM; you cannot define an identifier identical to a keyword. CASM recognizes five types of keywords:

- instructions
- directives
- operators
- registers
- predefined numbers

Table 2-6 lists the CASM reserved words.

Table 2-6. Reserved Words

Predefined Numbers

BYTE	WORD	LONG	DWORD	PWORD
QWORD	TWORD			

Operators

AND	EQ	GE	GT	HIGH
HIGHW	LAST	LE	LENGTH	LOW
LOWW	LT	MOD	NE	NOT
OFFSET	OR	SHL	SHR	TYPE
XOR				

Assembler Directives

ALIGN	BSS	CODE	DATA	DB
DW	DL	DD	DP	DQ
DT	#DEFINE	EJECT	ELSE	#ELSE
END	ENDIF	#ENDIF	EQU	EXTRN
IF	#IF	#IFDEF	IFLIST	INCLUDE
LIST	NOIFLIST	NOLIST	ORG	PAGESIZE
PAGEWIDTH	PUBLIC	RB	RD	RL
RP	RQ	RT	RW	SECTION
SET	SIMFORM	TITLE	USE16	USE32

Register Keywords

EAX	EBX	ECX	EDX	EBP	ESP	ESI	EDI
AX	BX	CX	DX	BP	SP	SI	DI
AH	AL	BH	BL	CH	CL	DH	DL
DS	SS	ES	FS	GS	EFLAGS	EIP	

Numeric Data Processor Registers

ST	ST0	ST1	ST2	ST3
	ST4	ST5	ST6	ST7

Default Section Names

CODE	DATA	BSS

Section 2.11 lists the 80386 instruction mnemonic keywords and the actions they initiate; Section 3 discusses CASM directives, and Section 2.8 defines operators.

2.7.2 Symbol Identifiers

A symbol is a user-defined identifier with attributes specifying the kind of information the symbol represents. Symbols fall into three categories:

- variables
- labels
- numbers

Variables

Variables identify data stored at a particular location in memory. All variables have two attributes:

Offset — determines the number of bytes between the beginning of the section and the location of the variable. The offset of a variable is the address of the variable relative to the starting address of the section. The offset is subject to relocation at link time. (See Appendix A).

Type — determines the number of bytes of data manipulated when the variable is referenced. A variable has one of the following type attributes:

BYTE	WORD	LONG	DWORD
PWORD	QWORD	TWORD	

The data definition directives define a variable as one of these types (see Section 3). For example, the variable, `my_variable`, is defined when it appears as the name for a data definition directive:

```
my_variable db 0
```

You can also define a variable as the name for an EQU directive referencing another variable, as shown in the following example:

```
another_variable equ my_variable
```

Labels

Labels identify locations in memory containing instruction statements. They are referenced with jumps or calls. All labels have two attributes: segment and offset. Label segment and offset attributes are essentially the same as variable segment and offset attributes. A label is defined when it precedes an instruction. A colon separates the label from instruction. For example,

```
my_label:   add  ax,bx
```

A label can also appear as the name for an EQU directive referencing another label. For example,

```
another_label   equ   my_label
```

Labels can also appear on lines without instruction mnemonics.

Numbers

You can also define numbers as symbols. CASM treats a number symbol as though you have explicitly coded the number it represents. For example,

```
number_five     equ    5
mov    al,number_five
```

is equivalent to the following:

```
mov    al,5
```

Section 2.8 describes operators and their effects on numbers and number symbols.

2.7.3 Example Identifiers

The following are valid identifiers:

```
NOLIST
WORD
AH
Mean_streets
crashed
variable number 1234567890
```

2.8 OPERATORS

CASM operators define the operations forming the values used in the final assembly instruction.

CASM operators fall into the following categories:

- arithmetic
- logical
- relational
- segment override
- variable manipulation
- variable creation

The following subsections define the operators in detail. Where the syntax of the operator is illustrated, **a** and **b** represent two elements of the expression. Unless otherwise specified, a and b represent absolute numbers, such as numeric constants, whose value is known at assembly-time. A relocatable number, on the other hand, is a number whose value is unknown at assembly-time, because it can change during the linking process. For example, the offset of a variable located in a segment that will be combined with some other segments at link-time is a relocatable number.

Table 2-8 on page 2-16 summarizes the CASM operators.

2.8.1 Arithmetic Operators

Addition and Subtraction

Addition and subtraction operators compute the arithmetic sum and difference of two operands. The first operand (**a**) can be a variable, label, an absolute number, or a relocatable number. For addition, the second operand (**b**) must be a number. For subtraction, the second operand can be a number, or it can be a variable or label in the same segment as the first operand.

When a number is added to a variable or label, the result is a variable or label with an offset whose numeric value is the second operand plus the offset of the first operand. Subtraction from a variable or label returns a variable or label whose offset is the first operand's offset, decremented by the number specified in the second operand.

Syntax:

a + b	returns the sum of **a** and **b**. Where **a** is a variable, label, absolute number, or relocatable number.
a - b	returns the difference of **a** and **b**. Where **a** and **b** are variables, labels, absolute numbers, or relocatable numbers in the same segment.

Example:

```
count   equ   2
displ   equ   5
flag    db    offh
  .
  .
  .
mov   al,flag+1
mov   cl,flag+displ
mov   bl,displ-count
```

Multiplication and Division

The multiplication and division operators *, /, and MOD accept only numbers as operands. * and / treat all operators as unsigned numbers.

Syntax:

a * b	unsigned multiplication of **a** and **b**
a / b	unsigned division of **a** and **b**
a MOD b	return remainder of **a** / **b**

Example:

```
mask    equ   0fch
signbit equ   80h
mov   cl,mask and signbit
mov   al,not mask
```

2.8.5 Relational Operators

Relational operators treat all operands as unsigned numbers. The relational operators are EQ (equal), LT (less than), LE (less than or equal), GT (greater than), GE (greater than or equal), and NE (not equal). Each operator compares two operands and returns all ones (0FFFFH) if the specified relation is true, and all zeros if it is not.

Syntax:

In all of the operators below, **a** and **b** are unsigned numbers; or they are labels, variables, or relocatable numbers defined in the same segment.

a EQ b	returns 0FFFFH if **a** = **b** otherwise 0
a LT b	returns 0FFFFH if **a** < **b**, otherwise 0
a LE b	returns 0FFFFH if **a** <= **b**, otherwise 0
a GT b	returns 0FFFFH if **a** > **b**, otherwise 0
a GE b	returns 0FFFFH if **a** >= **b**, otherwise 0
a NE b	returns 0FFFFH if **a**<> **b**, otherwise 0

Example:

```
limit1  equ   10
limit2  equ   25
mov   ax,limit1 lt limit2
mov   ax,limit1 gt limit2
```

2.8.6 Segment Override Operator

When manipulating variables, CASM decides which segment register to use. You can override this choice by specifying a different register with the segment override operator.

Note: The programming model supported by FlexOS (and COFF) is non-segmented, so use of the segment override should be unnecessary.

Syntax:

seg:	overrides segment register selected by assembler. **seg** can be: CS, DS, ES, FS, GS, or SS.

2.8.2 Shift Operators

The shift operators perform a bit-wise shift of the operand.

Syntax:

a SHL b	returns the value resulting from shifting **a** to left by the amount specified by **b**
a SHR b	returns the value resulting from shifting **a** to the right by an the amount specified by **b**

Example:

```
shl     ax,1
shl    eax,17
```

2.8.3 Unary Operators

Unary operators specify a number as either positive or negative. CASM unary operators accept both signed and unsigned numbers.

Syntax:

+ a	gives **a**
- a	gives 0 - **a**

Example:

```
mov    cl,+35
mov    al,2--5
mov    dl,-12
```

2.8.4 Logical Operators

Logical operators accept only numbers as operands. They perform the Boolean logic operations AND, OR, XOR, and NOT.

Syntax:

a XOR b	bit-by-bit logical EXCLUSIVE OR of **a** and **b**
a OR b	bit-by-bit logical OR of **a** and **b**
a AND b	bit-by-bit logical AND of **a** and **b**
NOT a	logical inverse of **a**: all 0s become 1s, 1s become 0s. (**a** is a 16-bit number.)

Example:

```
mov    ax,ss:wordbuffer[bx]
mov    cx,es:array
cs:movsb
```

2.8.7 Variable Manipulation Operators

A variable manipulator creates a number equal to one attribute of its variable operand. OFFSET extracts the variable's offset value; TYPE, its type value (1, 2, or 4), and LENGTH, the number of bytes associated with the variable. LAST compares the variable's LENGTH with zero. If LENGTH is greater than zero, LAST decrements LENGTH by one. If LENGTH equals zero, LAST leaves it unchanged. Variable manipulators accept only variables as operators.

Syntax:

OFFSET a	creates a number whose value is the offset value of the variable or label **a**.
TYPE a	creates a number. If the variable **a** is of type BYTE, WORD or LONG, the value of the number created is 1, 2, or 4, respectively.
LENGTH a	creates a number whose value is the length attribute of the variable **a**. The length attribute is the number of bytes associated with the variable.
LAST a	if LENGTH **a** > 0, then LAST **a** = LENGTH **a** - 1; if LENGTH **a** = 0, then LAST **a** = 0.

Example:

```
wordbuffer      dw     0,0,0
buffer          db     1,2,3,4,5
    .
    .
    .
mov    ax,length buffer
mov    ax,last buffer
mov    ax,type buffer
mov    ax,type wordbuffer
```

2.8.8 Variable Creation Operators

Three CASM operators are used to create variables. These are the period, dollar sign, and "type" operators described below.

The **period** operator (.) creates a variable from a number.

The **dollar sign** operator ($) creates a label with an offset attribute equal to the current value of the location counter. This operator takes no operand.

The **"type"** operator creates a virtual variable or label valid only during the execution of the instruction. The temporary symbol has the same Type attribute as the "type" operator, and all other attributes of the operand.

Syntax:

.a	creates variable with an offset attribute of **a**. Segment attribute is current data segment.
$	creates label with offset equal to current value of location counter; segment attribute is current segment.
"type" a	creates virtual variable or label with "type" and attributes of **a**. "type" can be a BYTE, WORD, DWORD, QWORD, PWORD, or TWORD; **a** is the address of the expression.

Examples:

```
mov   byte [bx], 5
mov   al,byte [bx]
inc   word [si]

mov   ax, .0
mov   bx, es:.4000h

jmp   $
jmps  $
jmp   $+3000h
```

2.8.9 Isolation Operators

The isolation operators return either the high or low portion of the operand.

Syntax:

HIGH a	returns the high-order byte of **a**, which can be a 16-bit (or greater) sized operand
LOW a	returns the low-order byte of **a**, which can be a 16-bit (or greater) sized operand
HIGHW a	returns the high-order word of **a**, which can be a 32-bit (or greater) sized operand
LOWW a	returns the low-order word of **a**, which can be a 32-bit (or greater) sized operand

Table 2-8. CASM Operator Summary

Operator	Description
+	addition or unary positive
-	subtraction or unary negative
*	multiplication
/	unsigned division
MOD	return remainder of division
AND	logical AND
NOT	logical NOT
OR	logical OR
XOR	logical eXclusive OR
SHR	shift right
SHL	shift left
EQ	Equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
NE	Not Equal to
.	create variable, assign offset
$	create label, offset = location counter
LAST	compare LENGTH of variable to 0
LENGTH	create number from variable length
OFFSET	create number from variable offset
seg:addr	override segment register
TYPE	create number from variable type
"type"	form a variable of type "type", (BYTE, WORD, etc)
LOW	return low-order byte of 16-bit or greater sized operand
LOWW	return low-order half-word of 32-bit or greater sized operand
HIGH	return high-order byte of 16-bit or greater sized operand
HIGHW	return high-order half-word of 32-bit or greater sized operand

2.9 EXPRESSIONS

CASM allows address, numeric, and bracketed expressions. An address expression evaluates to a memory address and has two components:

- offset value
- type

Both variables and labels are address expressions. An address expression is not a number, but its components are numbers. You can combine numbers with operators to make an address expression.

Examples:

Given the definition

```
foobar  equ  012345678h
```

the following instructions are equivalent:

```
mov dx, highw(foobar)     mov  dx,1234h
mov cx, highw(foobar)     mov  cx,5678h
```

2.8.10 Operator Precedence

Expressions combine variables, labels, or numbers with operators. CASM allows several kinds of expressions (see Section 2.9). This section defines the order that CASM performs operations if more than one operator appear in an expression.

CASM evaluates expressions from left to right, but evaluates operators with higher precedence before operators with lower precedence. When two operators have equal precedence, CASM evaluates the leftmost operator first. Table 2–7 shows CASM operators in order of increasing precedence.

You can use parentheses to override the precedence rules. CASM first evaluates the part of an expression enclosed in parentheses. If you nest parentheses, CASM evaluates the innermost expressions first. For example,

```
15/3 + 18/9 = 5 + 2 = 7
15/(3 + 18/9) = 15/(3 + 2) = 15/5 = 3
```

Note that CASM allows five levels of nested parentheses.

Table 2–7. Precedence of Operators

Order (1 = highest)	Operator
1	XOR, OR
2	AND
3	NOT
4	EQ, LT, LE, GT, GE, NE
5	+, -
6	*, /, MOD, SHL, SHR
7	HIGH, HIGHW, LOW, LOWW
8	+, -
9	segment_override:
10	SEG, OFFSET, TYPE, LENGTH, LAST
11	( ), [ ]
12	., $

A numeric expression evaluates to a number. It contains no variables or labels, only numbers and operands.

Bracketed expressions specify base- and index-addressing modes. For example,

Mode	Example
Direct	`mov EAX,word_var`
Register Indirect	`mov SI,[EBX]`
Based or Indexed	`mov 4[ECX],EAX`
Based-Indexed	`mov AL,10[BX+DX]`
Scaled	`mov SI,[EBX*4]` or
	`mov 4[ECX*2],EAX`

Memory addressing syntax such as the following is not permitted:

```
mov AL,10 + [BX] + 8 * [DX]
```

although under other assemblers, it may have signified the same mode as

```
mov  AL,10[BX+DX*8]
```

For a complete explanation of memory addressing modes, see the Intel 80386 Programmer's Reference Manual.

2.10 STATEMENTS

Statements can be instructions or directives. CASM translates instructions into 80386 machine language instructions. CASM does not translate directives into machine code. Directives tell CASM to perform certain functions.

You must terminate each assembly language statement with a carriage return (CR) and line-feed (LF), or exclamation point. CASM treats these as an end-of-line. You can write multiple assembly language statements without comments on the same physical line and separate them with exclamation points. Only the last statement on a line can have a comment because the comment field extends to the physical end of the line.

Instruction statements have the following syntax:

```
[label:] [pre1] [pre2] [pre3] [pre4] mnemonic [operand(s)] [;comment]
```

The fields are defined as follows:

label	A symbol followed by a colon defines a label at the current value of the location counter in the current segment. This field is optional.
pre1, etc.	A prefix such as ASP, OSP, REP, CS:, or LOCK. This field is optional.

mnemonic — A symbol defined as a machine instruction, either by CASM or by an EQU directive. This field is optional unless preceded by a prefix instruction. If you omit this field, no operands can be present, although the other fields can appear. Section 2.11 lists the CASM mnemonics.

operand(s) — An instruction mnemonic can require other symbols to represent operands to the instruction. Instructions can have zero, one, or two operands.

comment — Any semicolon appearing outside a character string begins a comment. A comment ends with a carriage return. This field is optional, but you should use comments to facilitate program maintenance and debugging.

Note: Labels and comments are allowed to exist on a line without the presence of a mnemonic.

Section 3 describes the CASM directives.

2.11 INSTRUCTION SET SUMMARY

Table 2-9 summarizes the complete CASM instruction set in alphabetical order. For a more detailed description of each instruction, including bit patterns, see the Intel 80386 Programmer's Reference Manual.

Table 2-9. CASM Instruction Summary

Mnemonic	Description
AAA	ASCII adjust after addition
AAD	ASCII adjust AX before division
AAM	ASCII adjust AX after multiplication
AAS	ASCII adjust AL after subtraction
ADC	Add with carry
ADD	Add
AND	Logical And
ARPL	Adjust priviledge level
ASP	Address size prefix
BOUND	Check array index against bounds
BSF	Bit scan forward
BSR	Bit scan reverse
BT	Bit test
BTC	Bit test and complement
BTR	Bit test and reset
BTS	Bit test and set

Table 2-9. (Continued)

Mnemonic	Description
FFREE	Free register
FIADD	Integer add
FICOM	Integer compare
FICOMP	Integer compare and pop
FIDIV	Integer divide
FIDIVR	Integer divide reversed
FILD	Integer load
FIMUL	Integer multiply
FINCSTP	Increment stack pointer
FINIT/FNINIT	Initialize processor
FIST	Integer store
FISTP	Interger store and pop
FISUB	Integer subtract
FISUBR	Integer subtract reversed
FLD	Load Real
FLDCW	Load control word
FLDENV	Load environment
FLDPI	Load 80-bit value for pi.
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$
FLDZ	Load + 0.0
FLD1	Load + 1.0
FMUL	Multiply real
FMULP	Multiply real and pop
FNOP	No operation
FPATAN	Partial arctangent
FPREM	Partial remainder
FPREM1	Partial remainder (IEEE)
FPTAN	Partial tangent
FRNDINT	Round to integer
FRSTOR	Restore state
FSAVE/FNSAVE	Save state
FSCALE	Scale
FSIN	Sine of ST(0)
FSINCOS	Sine and Cosine of ST(0)
FST	Store Real
FSTP	Store Real and pop
FSTENV/FNSTENV	Store environment
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word

Table 2-9. (Continued)

Mnemonic	Description
CALL	Call (intra segment)
CALLF	Call (inter segment)
CBW	Convert byte to word
CDQ	Convert dword to qword
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CLTS	Clear TS flag
CMC	Complement carry flag
CMP	Compare
CMPSB	Compare byte (of string)
CMPSD	Compare dword (of string)
CMPSL	Compare long (of string)
CMPSW	Compare word (of string)
CWD	Convert word to dword
CWDE	Convert word to dword extended
DAA	Decimal adjust AL after addition
DAS	Decimal adjust AL after subtraction
DEC	Decrement by 1
DIV	Divide (unsigned)
ENTER	Procedure entry
ESC	Escape
F2XM1	2^x-1
FABS	Absolute value
FADD	Add Real
FADDP	Add Real and pop
FBLD	Packed Decimal load
FBSTP	Packed Decimal store and pop
FCHS	Change sign
FCLEX/FNCLEX	Clear exceptions
FCOM	Compare Real
FCOMP	Compare Real and pop
FCOMPP	Compare Real and pop twice
FCOS	Cosine of ST(0)
FDECSTP	Decrement stack pointer
FDISI/FNDISI	Disable interrupts
FDIV	Divide Real
FDIVP	Divide Real and pop
FDIVR	Divide Real reversed
FDIVRP	Divide Real reversed and pop
FENI/FNENI	Enable interrupts

Table 2-9. (Continued)

Mnemonic	Description
JMPS	Jump (8 bit displacement)
JNA	Jump on not above
JNAE	Jump on not above or equal
JNB	Jump on not below
JNBE	Jump on not below or equal
JNC	Jump on not carry
JNE	Jump on not equal
JNG	Jump on not greater
JNGE	Jump on not greater or equal
JNL	Jump on not less
JNLE	Jump on not less or equal
JNO	Jump on not overflow
JNP	Jump on not parity
JNS	Jump on not sign
JNZ	Jump on not zero
JO	Jump on overflow
JP	Jump on parity
JPE	Jump on parity even
JPO	Jump on parity odd
JS	Jump on sign
JZ	Jump on zero
LAHF	Load AH with flags
LAR	Load Access rights
LDS	Load Pointer into DS
LEA	Load effective address
LEAVE	High level procedure exit
LES	Load pointer into ES
LFS	Load pointer into FS
LGDT	Load Global Descriptor Table register
LGS	Load pointer into GS
LIDT	Load Interrupt Descriptor Table register
LLDT	Load Local Descriptor Table register
LMSW	Load machine status word
LOCK	Lock bus
LODSB	Load byte (of string)
LODSD	Load dword (of string)
LODSL	Load long (of string)
LODSW	Load word (of string)
LOOP	Loop
LOOPE	Loop while equal
LOOPNE	Loop while not equal
LOOPNZ	Loop while not zero

Table 2-9. (Continued)

Mnemonic	Description
FSQRT	Square root
FSUB	Subtract Real
FSUBP	Subtract Real and pop
FSUBR	Subtract Real reversed
FSUBRP	Subtract Real reversed and pop
FTST	Test
FUCOM	Unordered compare
FUCOMP	Unordered compare and pop
FUCOMPP	Unordered compare and pop
FWAIT	CPU wait
FXAM	Examine
FXCH	Exchange registers
FXTRACT	Extract exponent and significand
FYL2X	$Y * \log_2 X$
FYL2XP1	$Y * \log_2(X + 1)$
HLT	Halt
IBTS	Insert bit string
IDIV	Integer divide (signed)
IMUL	Integer multiply (signed)
IN	Input byte or word from port
INC	Increment by 1
INSB	Input byte from port to string
INSD	Input dword from port to string
INSL	Input long from port to string
INSW	Input word from port to string
INT	Interrupt
INTO	Interrupt on overflow
IRET\IRETD	Interrupt return
JA	Jump on above
JAE	Jump on above or equal
JB	Jump on below
JBE	Jump on below or equal
JC	Jump on carry
JCXZ	Jump on CX zero
JE	Jump on equal
JECXZ	Jump on ECX zero
JG	Jump on greater
JGE	Jump on greater or equal
JL	Jump on less
JLE	Jump on less or equal
JMP	Jump (intra segment)
JMPF	Jump (inter segment)

Table 2-9. (Continued)

Mnemonic	Description
LOOPZ	Loop while zero
LSL	Load segment limit
LSS	Load pointer into SS
LTR	Load task register
MOV	Move
MOVSB	Move byte (of string)
MOVSD	Move dword (of string)
MOVSL	Move long (of string)
MOVSW	Move word (of string)
MOVSX	Move with sign-extend
MOVZX	Move with zero-extend
MUL	Multiply
NEG	Two's complement negate
NOP	No Operation
NOT	One's complement negate
OR	Logical inclusive OR
OSP	Operand size prefix
OUT	Output byte or word pointer [si] to DX
OUTSB	Output byte pointer [si] to DX
OUTSD	Output dword pointer [si] to DX
OUTSL	Output long pointer [si] to DX
OUTSW	Output word pointer [si] to DX
POP	Pop a word from stack
POPA\POPAD	Pop all general registers
POPF\POPFD	Pop stack into FLAGS or EFLAGS register
PUSH	Push operand on stack
PUSHA\PUSHAD	Push all general registers
PUSHF\PUSHFD	Push FLAGS register onto stack
RCL	Rotate through carry left
RCR	Rotate through carry right
REP	Repeat
REPE	Repeat while equal
REPNE	Repeat while not equal
REPNZ	Repeat while not zero
REPZ	Repeat while zero
RET	Return (intra segment)
RETF	Return (inter segment)
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into FLAGS
SAL	Shift arithmetic left
SAR	Shift arithmetic right

Table 2-9. (Continued)

Mnemonic	Description
SBB	Integer subtract with borrow
SCASB	Scan byte (of string)
SCASD	Scan dword (of string)
SCASL	Scan long (of string)
SCASW	Scan word (of string)
SETA	Set byte if above
SETAE	Set byte if above or equal
SETB	Set byte if below
SETBE	Set byte if below or equal
SETC	Set byte if carry
SETE	Set byte if equal
SETG	Set byte if greater than
SETGE	Set byte if greater than or equal
SETL	Set byte if less than
SETLE	Set byte if less than or equal
SETNA	Set byte if not above
SETNAE	Set byte if not above or equal
SETNB	Set byte if not below
SETNBE	Set byte if not below or equal
SETNC	Set byte if not carry
SETNE	Set byte if not equal
SETNG	Set byte if not greater
SETNGE	Set byte if not greater than or equal
SETNL	Set byte if not less than
SETNLE	Set byte if not less than or equal
SETNO	Set byte if not overflow
SETNP	Set byte if not parity
SETNS	Set byte if not sign
SETNZ	Set byte if not zero
SETO	Set byte if overflow
SETP	Set byte if parity
SETPE	Set byte if parity even
SETPO	Set byte if parity odd
SETS	Set byte if sign
SETZ	Set byte if zero
SGDT\SGDTE	Store Global Descriptor Table register ??
SHL	Shift left
SHR	Shift right
SHLD	Double precision shift left
SHRD	Double precision shift right
SIDT	Store Interrupt Descriptor Table register
SLDT	Store Local Descriptor Table register

Table 2-9. (Continued)

Mnemonic	Description
SMSW	Store machine Status word
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOSB	Store byte (of string)
STOSD	Store dword (of string)
STOSL	Store long (of string)
STOSW	Store word (of string)
STR	Store Task register
SUB	Integer subtract
TEST	Test (logical compare)
VERR	Verify read access
VERW	Verify write access
WAIT	Wait until BUSY# pin is inactive
XCHG	Exchange register/memory with register
XBTS	Extract bit string
XLAT\XLATB	Translate
XOR	Exclusive OR

2.12 FLAGS

The 80386 EFLAGS register contains a set of flags that reflect the state of the processor. Table 2-10 lists the flags you can test to determine the effects of an executed instruction upon an operand or register.

Table 2-10. Flag Register Symbols

Symbol	Meaning
AF	Auxiliary Carry Flag
CF	Carry Flag
DF	Direction Flag
IF	Interrupt Enable Flag
IOPL	I/O Priveledge Level Field
NF	Nested Flag
OF	Overflow Flag
PF	Parity Flag
RF	Resume Flag
SF	Sign Flag
TF	Trap Flag
VM	Virtual Mode Flag
ZF	Zero Flag

Table 2-11 summarizes the effects of arithmetic instructions on flag bits.

Table 2-11. Effects of Arithmetic Instructions on Status Flags

Flag Bit	Result
CF	is set if the operation results in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; otherwise CF is cleared.
AF	is set if the operation results in a carry out of (from addition) or a borrow into (from subtraction) the low-order four bits of the result; otherwise AF is cleared.
ZF	is set if the result of the operation is zero; otherwise ZF is cleared.
SF	is set if the result is negative.
PF	is set if the modulo 2 sum of the low-order eight bits of the result of the operation is 0 (even parity); otherwise PF is cleared (odd parity).
OF	is set if the operation results in an overflow; the size of the result exceeds the capacity of its destination.

2.13 16/32 BIT OPERANDS AND ADDRESSES

When the 80386 runs in protected mode, a bit in the code segment descriptor (D-bit) determines how the CPU decodes instructions. The same opcode and addressing mode byte can specify an entirely different operation depending on the state of this bit. Generally, if the bit is set a 32-bit operation is performed, else a 16-bit operation. A 32-bit operation implies 32-bit data manipulation and 32-bit address formation. The following example illustrates this concept:

```
39h 07h = CMP [BX],AX      if in 16-bit mode, or
          CMP [EDI],EAX    if in 32-bit mode.
```

CASM provides two instruction prefixes to override the default operation size specified by the D-bit. If the D-bit is set, the presence of an **Operand Size Prefix (OSP)** allows movement or manipulation of 16-bit registers and data. If the D-bit is reset, an OSP indicates 32-bit data.

An **Address Size Prefix (ASP)** reverses the default address-formation mode. If the D-bit is set, the presence of an ASP indicates 16-bit addressing. If it is reset, 32-bit.

CASM assembles instructions using 16 and/or 32 bit operands and addressing by inserting the appropriate Size Prefixes if required, using the type and sizes of the instruction operands to determine the need for the prefixes.

CASM also provides two directives (pseudo-ops) to specify which default operation is intended to be in effect when the assembled code is executed.

The **USE32** directive instructs CASM to insert one or both size prefixes before the assembled instruction when a 16-bit operation is encountered. No prefixes are required for 32-bit instructions. The **USE16** directive works in the opposite way. Prefixes are inserted if 32-bit instructions are encountered.

Normally, only one USE directive is used at the start of a single module.

2.14 PREFIXES AND OVERRIDES

Depending on the assembly size in effect (determined by the USE16/USE32 directives) when processing any given instruction, CASM may insert an Operand Size Prefix (OSP) and/or an Address Size Prefix (ASP) before the primary opcode. By default, the assembly mode is USE32, which corresponds to the default setting of the "Operation Size" bit in a .386 program's code segment descriptor. CASM also allows all prefixes, including those above, to be explicitly specified in the source file.

A maximum of 4 prefixes may precede the instruction. They may be entered in any order, but CASM uses the following order when building the instruction:

1st	2nd	3rd	4th
LOCK (F0h)	ASP (67h)	OSP (66h)	CS: (2Eh)
REP (F3h)			DS: (3Eh)
REPE/REPZ (F3h)			ES: (26h)
REPNE/REPNZ (F2h)			SS: (36h)
			FS: (64h)
			GS: (65h)

No more than one from each of the four groups is permitted for a single instruction.

2.15 JUMP OPTIMIZATION

A short jump (+- 128 bytes) may be explicitly coded by using the JMPS mnemonic. However, by default CASM converts a JMP instruction to a short jump assuming the target is not external, is close enough, and the command-line specification, -J, has not been entered. You can also use the JMPF mnemonic, but is not required because CASM can determine the type of jump by the operand and the 16/32 assembly mode. Therefore, you normally need to supply only one form of the jump instruction, JMP.

You can save one byte in an immediate near jump by specifying only a 16-bit displacement. Assuming a USE32 code section and a 16-bit displacement, CASM would insert an ASP prefix, then supply a 16-bit displacement instead of 32-bits.

2.16 INTER-SEGMENT CONTROL TRANSFERS

CASM (and the COFF format) supports the "flat", or unsegmented memory addressing scheme of the 80386 microprocessor running in **protected** mode. This has important implications for use of the inter-segment control transfer instructions, JMPF and CALLF.

FlexOS's memory management scheme assumes that a 386 application does not alter its segment registers. An SVC call is performed by an INT instruction, which means that calls and/or jumps to fixed logical addresses are not required. A 386 application under FLEXOS need never perform such transfers.

Other protected operating environments may use fixed segmented addresses, supported by the 80386 in the form of "call gates", to allow interprogram transfers. CASM supports both forms of intersegment transfers, immediate and indirect, when assembling CALLF and JUMPF instructions.

An immediate transfer is coded as follows:

```
MONITOR_ENTRY equ 1067:00001024h
callf MONITOR_ENTRY
```

An indirect transfer is coded as follows:

```
SV_CALL equ 0f7:00039ab0h
code
callf vector
    .
    .
data
vector dp MONITOR_ENTRY
```

Note: CLINK cannot resolve immediate intersegment transfers to symbols external to the source module.

2.17 AMBIGUOUS INSTRUCTIONS

On the Intel 386 processor, using 8088, 8086, and 80286 mnemonics can in some cases lead to potential operand/address size ambiguities. For example, on the 8088/8086/80286 processors, the opcode A7h is the **CMPSW** instruction. It is a fixed-operand instruction, i.e., the word pointed to by ES:DI is always compared to the word pointed to by DS:SI. (A segment override is allowed.) The instruction performs identically when executing in a 16-bit segment on the 386. However, in a 32-bit segment the 32-bit registers, EDI and ESI are used as pointers and 32-bit data, longs are being operated on.

Using standard Intel mnemonics, CMPSW and CMPSD map to the same opcode. The question then becomes how to compare 16-bit data when in a 32-bit segment and vice-versa, and how to similarly use 16-bit addressing? CASM supports two means for resolving the ambiguity.

The first is to manually code an ASP and/or an OSP before the CMPSW/D instruction. The second requires the presence of 1 or 2 operands to tell CASM which prefixes to insert.

For example, the following are legal uses of the CMPSW/D instruction:

```
                ;Code:          Meaning: "compare..."
USE16
cmpsw           ;A7             ... the words at ES:[ DI] and DS:[ SI]
asp cmpsw       ;67 A7          ... the words at ES:[EDI] and DS:[ESI]
asp cmpsd       ;67 66 A7       ... the longs at ES:[EDI] and DS:[ESI]
cmpsd           ;66 A7          ... the longs at ES:[ DI] and DS:[ SI]

cmpsw [ DI],[ SI] ;A7           ... the words at ES:[ DI] and DS:[ SI]
cmpsw [EDI],[ESI] ;67 A7        ... the words at ES:[EDI] and DS:[ESI]
cmpsd [EDI],[ESI] ;67 66 A7     ... the longs at ES:[EDI] and DS:[EDI]
cmpsd [ DI],[ SI] ;66 A7        ... the longs at ES:[ DI] and DS:[ DI]
```

The following are examples of other similar instructions:

```
LODSB           LODSB [SI]      STOSB           STOSB [DI]
ASP LODSB       LODSB [ESI]     ASP STOSB       STOSB [EDI]
LODSW           LODSW [SI]      STOSW           STOSW [DI]
ASP LODSW       LODSW [ESI]     ASP STOSW       STOSW [EDI]
LODSD           LODSD [SI]      STOSD           STOSD [DI]
ASP LODSD       LODSD [ESI]     ASP STOSB       STOSD [EDI]

Similarly for:

MOVSW , MOVSW [DI],[SI] , ASP MOVSW , MOVSW [EDI],[ESI]
CMPSW , CMPSW [DI],[SI] , ASP CMPSW , CMPSW [EDI],[ESI]
SCASW , SCASW [DI]      , ASP SCASW , SCASW [EDI]

OUTSW , OUTSW [SI]      , ASP OUTSW , OUTSW [ESI]
INSW  , INSW  [SI]      , ASP INSW  , INSW  [ESI]

LOOP lab , ASP LOOP lab, LOOP CX,lab , LOOP ECX,lab

XLAT  , ASP XLAT , XLAT BX,  XLAT EBX

REP (the addressing mode in the instruction being repeated
 determines whether CX or ECX is used... e.g.,
 REP MOVSW [EDI],[ESI])
```

End of Section 2

CASM DIRECTIVES

CASM directives control various aspects of the assembly process. Directives are grouped into the following categories:

- code generation
- section control
- linkage control
- conditional assembly
- symbol definition
- data definition and memory allocation
- output listing control
- miscellaneous

3.1 DIRECTIVE SYNTAX

Directives have the following general syntax:

[name] directive operand(s) [;comment]

The fields are defined as follows:

name	Is a symbol that retains the value assigned by the directive. A name is required for the EQU directive, but it is optional for the other directives. Unlike the label field of an instruction, the name field of a directive is never terminated with a colon.
directive	One of the directive keywords defined in this section.
operand(s)	Analogous to the operands for instruction mnemonics. Some directives, such as DB and DW allow any operand; others have special requirements.
comment	Exactly as defined for instruction statements in Section 2.10.

The following sections describe each CASM directive. The syntax for each directive follows each section heading.

3.2 CODE GENERATION DIRECTIVES

Code generation directives control the type of code output by CASM. The code generation directives are:

- USE16/32
- ALIGN

3.2.1 USE16/32 Directive

Syntax: USE16
USE32

The USE16/32 directive establishes the default operand-size and addressing-size for subsequent instructions.

CASM inserts 16-32 bit mode-switching prefixes (ASPs and/or OSPs) preceding subsequent instructions that would otherwise conflict with the default mode.

Examples:

```
     USE16
     mov  EAX,4          ; An OSP will be inserted before this
                         ; instruction...
     mov  AX,4           ; but not this one.

; The 1st 3 bytes of the last two instructions are identical.
; The length of the last instruction is 3 bytes, the 1st is 5 bytes (w/out
; prefix).

               -------------------

     USE16
     mov  [EBX],AX       ; An ASP will be inserted before this instruction
                         ; because the addressing size is 32-bits, which
                         ; conflicts with the default.  No OSP will
                         ; be inserted because the operand size in the
                         ; instruction matches the default operand size
                         ; indicated by the USE16 directive.

               -------------------

     USE32
     mov  al,4
     mov  AX,4           ; An OSP will be inserted before this
                         ; instruction...
     mov  EAX,4          ; but not this one.

               -------------------
```

3.2.2 ALIGN Directive

Syntax: ALIGN expression

The ALIGN directive tells CASM to adjust the program counter to an address which is the next multiple of **expression**. It is most often used as ALIGN WORD or ALIGN LONG.

3.3 SECTION CONTROL DIRECTIVES

Section control directives affect the placement of sections in the output file. The section control directives are:

- CODE
- DATA
- BSS
- SECTION

3.3.1 CODE

Syntax: CODE

The CODE directive tells CASM to place all subsequent output into the ".text" section of the output file. When CASM encounters this directive, it is functionally equivalent to SECTION '.text'.

The CODE directive remains in effect until a different section control directive is encountered, or if there is no other section control directive in the source stream.

3.3.2 DATA

Syntax: DATA

The DATA directive tells CASM to place all subsequent output into the ".data" section of the output file. When CASM encounters this directive, it is functionally equivalent to SECTION '.data'.

The DATA directive remains in effect until a different section control directive is encountered, or if there is no other section control directive in the source stream.

3.3.3 BSS

Syntax: BSS

The BSS directive specifies that all subsequent symbols and symbol types are identified as existing in the uninitialized "bss" section of the object file. The BSS directive remains in effect until another section control directive is encountered.

Note: CASM does not place the actual initialized data into the BSS section of the object file.

3.3.4 SECTION

Syntax: SECTION 'name' [numeric_expression]

The SECTION directive specifies all subsequent output goes to the object file section given by **name**, which is an ASCII string no longer than eight characters. CASM places the optional **numeric_expression** into the **s_flags** field of the section header in the COFF file. The SECTION directive remains in effect until another section control directive is encountered.

Normal values for numeric_expression are:

20H = STYP_TEXT (executable code)
40H = STYP_DATA (initialized data)
80H = STYP_BSS (uninitialized data)

CLINK uses the section name and the s_flags field to properly locate sections at link time.

Note: CASM allows a maximum of 9 named sections other than CODE, DATA, and BSS in a single module.

3.4 LINKAGE CONTROL DIRECTIVES

Linkage control directives modify the link process. The linkage control directives are:

- PUBLIC
- EXTRN
- END

3.4.1 PUBLIC

Syntax: PUBLIC name1[, name2, ...]

The PUBLIC directive is used to specify that the entry points and data enumerated are to be marked as global in the object file. This allows other linked modules to access these procedures or variables.

3.4.2 EXTRN

Syntax: EXTRN name1:type,name2:type, ...

The EXTRN directive tells CASM that the listed symbols are defined in other modules and are not defined in the module being assembled. For data variables, a type may be entered if CASM is to perform type-checking. If no type is present, no type checking is performed. External constants must be specified by an ABS type.

Allowable data types are:

- BYTE(8), WORD(16), LONG(32)
- DWORD(16:16), PWORD(16:32)
- QWORD(NDP real 64-bit), TWORD(NDP real 80-bit)

Allowable constant types are: ABS, ABS8, ABS16, ABS32.

3.4.3 END

Syntax: END [name]

The END directive informs CASM that no further processing is to be done in the current source file. If the "name" field is present, CASM marks the location specified by name (typically name is a label) as the entry point to this module. If LK386 attempts to link modules together to form an executable image and encounters entry points specified in 2 or more modules, it treats the condition as a non-recoverable error.

3.5 CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives are used to set up conditions controlling the instruction sequence. The conditional assembly directives are:

- IF
- ELSE
- ENDIF

3.5.1 IF, ELSE, and ENDIF Directives

Syntax:
```
IF numeric_expression
   source_line 1
   source_line 2
     .
     .
     .
   source_line n

ELSE
   alternate_source_line 1
   alternate_source_line 2
     .
     .
     .
   alternate_source_line n

ENDIF
```

The IF and ENDIF directives allow you to conditionally include or exclude a group of source lines from the assembly. The optional ELSE directive allows you to specify an alternative set of source lines. You can use these conditional directives to assemble several different versions of a single source program. You can nest IF directives to five levels.

When CASM encounters an IF directive, it evaluates the **numeric_expression** following the IF keyword. You must define all elements in the numeric_expression before you use them in the IF directive. If the value of the expression is nonzero, then CASM assembles **source_line 1** through **source_line n**.

If the value of the expression is zero, then CASM lists all the lines, but does not assemble them. If the value of the expression is zero, and you specify an ELSE directive, then CASM assembles **alternative_source_line 1** through **alternative_source_line n**.

3.5.2 C Language Conditional Compilation Directives

CASM also supports the following C language conditional compilation directives:

- #DEFINE
- #ELSE
- #ENDIF
- #IF
- #IFDEF

CASM supports these directives for compatibility with C language definition modules. See the Metaware™ High C documentation for details about these directives.

3.6 SYMBOL DEFINITION DIRECTIVES

There are two symbol definition directives:

- EQU
- SET

3.6.1 EQU Directive

Syntax: symbol_name EQU numeric_expression
symbol_name EQU address_expression
symbol_name EQU register
symbol_name EQU instruction_mnemonic

The EQU (equate) directive assigns values and attributes to user-defined symbols. Do not put a colon after the **symbol_name**. Once you define a symbol, you cannot redefine the symbol with a subsequent EQU or another directive. You must also define any elements used in a **numeric_expression** or an address expression before using the EQU directive.

The first form of the EQU directive assigns a numeric value to the symbol. The second form assigns a memory address. The third form assigns a new name to a register. The fourth form defines a new instruction (sub)set. The following examples illustrate these four EQU forms.

```
FIVE    EQU     2*2+1
NEXT    EQU     BUFFER
COUNTER EQU     CX
MOVVV   EQU     MOV
MOVVV   AX,BX
```

Note: CASM implements the EQU directive as a string replacement function (like the C language #define statement). This means that a value must be defined before being used. It also means some constructs may work unexpectedly. For example, the following code:

```
var1 equ 6
var2 equ 2 + 3
mov  cx, var1-var2
```

moves into CX the value 7 (6 - 2 + 3) not 1 (6 - (2+3)).

3.6.2 SET Directive

Syntax: symbol_name SET numeric_expression
symbol_name SET address_expression
symbol_name SET register
symbol_name SET instruction_mnemonic

The SET directive is identical to the EQU directive except the symbol it defines may be redefined in the same source file using a subsequent SET directive.

3.7 DATA AND MEMORY DIRECTIVES

Data definition and memory allocation directives define the storage format used for a specified expression or constant. The available data definition and memory allocation directives are:

DB DW DD DL DP DQ DT
RB RW RD RL RP RQ RT

3.7.1 DB Directive

Syntax: [symbol] DB numeric_expression [,numeric_expression...]
[symbol] DB string_constant [,string_constant...]

The DB directive defines initialized storage areas in byte format. CASM evaluates **numeric_expressions** to 8-bit values and sequentially places them in the object file. CASM places **string_constants** in the object file according to the rules defined in Section 2.5.3. Note that CASM does not perform translation from lower- to uppercase within strings.

The DB directive is the only CASM statement that accepts a string constant longer than four bytes. You can add multiple expressions or constants, separated by commas, to the definition if it does not exceed the physical line length.

Use an optional symbol to reference the defined data area throughout the program. The symbol has three attributes:

- offset
- type
- length

The offset attribute determines the symbol's memory reference; the type attribute specifies single bytes, and the length attribute tells the number of bytes reserved.

The following listing shows examples of DB directives and the resulting hexadecimal values:

```
TEXT    DB      'FlexOS system',0
AA      DB      'a' + 80H
X       DB      1,2,3,4,5
        .
        .
        .
MOV     CX,LENGTH TEXT
```

3.7.2 DW Directive

Syntax: [symbol] DW numeric_expression [,numeric_expression...]
[symbol] DW string_constant [,string_constant...]

The DW directive initializes two-byte words of storage. The DW directive initializes storage the same way as the DB directive, except that each **numeric_expression**, or **string_constant** initializes two bytes of memory with the low-order byte stored first. The DW directive does not accept string constants longer than two characters.

The following are examples of DW directives:

```
CNTR    DW      0
JMPTAB  DW      SUBR1,SUBR2,SUBR3
DW      1,2,3,4,5,6
```

3.7.3 DL Directive

Syntax: [symbol] DL address_expression [,address_expression...]

The DL directive initializes four bytes of storage. The DL directive does not accept string constants longer than 4 characters.

3.7.4 DD Directive

Syntax: [symbol] DD address_expression [,address_expression...]

The DD directive initializes four bytes of storage. DD follows the same procedure as DB, except that the offset attribute of the address expression is stored in the two lower bytes and the segment attribute is stored in the two upper bytes. For example,

```
CSEG

LONG_JMPTAB     DD      ROUT1,ROUT2
                DD      ROUT3,ROUT4
```

3.7.5 DP Directive

Syntax: [symbol] DP address_expression [,address_expression...]

The DP directive initializes six bytes of storage.

3.7.6 DQ Directive

Syntax: [symbol] DQ address_expression [,address_expression...]

The DQ directive initializes eight bytes of storage.

3.7.7 DT Directive

Syntax: [symbol] DT address_expression [,address_expression...]

The DT directive initializes ten bytes of storage.

3.7.8 RB Directive

Syntax: [symbol] RB numeric_expression

The RB directive allocates byte storage in memory without any initialization. The RB directive is identical to the RS directive except that it gives the byte attribute. For example,

```
BUF     RB      48
RB      4000H
RB      1
```

3.7.9 RW Directive

Syntax: [symbol] RW numeric_expression

The RW directive allocates two-byte word storage in memory but does not initialize it. The **numeric_expression** gives the number of words to be reserved. For example,

```
BUFF    RW      128
RW      4000H
RW      1
```

3.7.10 RL Directive

Syntax: [symbol] RL numeric_expression

The RL directive allocates four bytes of storage.

3.7.11 RD Directive

Syntax: [symbol] RD numeric_expression

The RD directive allocates four bytes of storage.

3.7.12 RP Directive

Syntax: [symbol] RP numeric_expression

The RP directive allocates six bytes of storage.

3.7.13 RQ Directive

Syntax: [symbol] RQ numeric_expression

The RQ directive allocates eight bytes of storage.

3.7.14 RT Directive

Syntax: [symbol] RT numeric_expression

The RT directive allocates ten bytes of storage.

3.8 LISTING CONTROL DIRECTIVES

Listing control directives modify the list file format. The listing control directives are:

EJECT	PAGEWIDTH
IFLIST\NOIFLIST	SIMFORM
LIST\NOLIST	TITLE
PAGESIZE	

3.8.1 EJECT Directive

Syntax: EJECT

The EJECT directive performs a page eject during printout. The EJECT directive is printed on the first line of the next page.

3.8.2 NOIFLIST/IFLIST Directives

Syntax: NOIFLIST
IFLIST

The NOIFLIST directive suppresses the printout of the contents of conditional assembly blocks that are not assembled. The IFLIST directive resumes printout of these blocks.

3.8.3 NOLIST\LIST Directives

Syntax: NOLIST
LIST

The NOLIST directive suppresses the printout of lines following the directive. The LIST directive restarts the listing.

3.8.4 PAGESIZE Directive

Syntax: PAGESIZE numeric_expression

The PAGESIZE directive defines the number of lines on each page. The default page size is 66 lines.

3.8.5 PAGEWIDTH Directive

Syntax: PAGEWIDTH numeric_expression

The PAGEWIDTH directive defines the number of columns printed across the page of the listing file. The default page width is 120 unless the listing is routed directly to the console; then the default page width is 79.

3.8.6 SIMFORM Directive

Syntax: SIMFORM

The SIMFORM directive replaces a form-feed (FF) character in the list file with the correct number of line-feeds (LF). Use this directive when directing a list file to a printer unable to interpret the form-feed character.

3.8.7 TITLE Directive

Syntax: TITLE string_constant

CASM prints the **string_constant** defined by a TITLE directive statement at the top of each printout page in the listing file. The title character string can be up to 30 characters in length. For example,

```
TITLE  'Serial driver'
```

3.9 MISCELLANEOUS DIRECTIVES

Additional CASM directives are:

- INCLUDE
- ORG

3.9.1 INCLUDE Directive

Syntax: INCLUDE filename

The INCLUDE directive includes another CASM source file in the source text. For example, to include the file EQUALS in your text, you would enter:

```
INCLUDE  EQUALS.A86
```

CASM first tries to open the file in the current directory. If CASM does not find the file in the current directory, it searches the directory containing the source file.

Note: You cannot nest INCLUDE directives; a source file called by an INCLUDE directive cannot contain another INCLUDE directive.

3.9.2 ORG Directive

Syntax: ORG numeric_expression

The ORG directive sets the offset of the location counter in the current section to a value specified by the **numeric_expression**. You must define all elements of the expression before using the ORG directive, and the expression must evaluate to an absolute number.

If you use an ORG statement in a section that CLINK does not combine with other sections at link-time, then the numeric_expression indicates the actual offset within the section.

If the section is combined with others at link-time, then numeric_expression is not an absolute offset. It is relative to the beginning address of the final section.

End of Section 3

CLINK - LINKAGE EDITOR

CLINK™ is the linkage editor that processes Common Object File Format (COFF) object files to produce executable files (filetype .386). There are three types of COFF object files:

- object files (filetype .O) from the CASM assembler or Metaware™ High C compiler
- library files (filetype .LIB) from CLIB
- shared run-time library files (type .LIB) created by CLINK.

4.1 CREATING THE .386 FILE

CLINK produces an executable (.386) file by making two passes through the input object file(s).

4.1.1 Pass 1

During pass 1, CLINK reads the program arguments, input files (see Section 4.3), object files and libraries. COFF section headers (see Appendix A) and global symbol information are taken from the object files and libraries. CLINK reads the section headers so that at the end of Pass 1, it knows what sections to place in the output file. CLINK reads the symbol information in order to search libraries for symbols that resolve undefined external references.

CLINK builds an internal **section structure** for any section that must be included in the output file and links it to the end of the list of input sections. There may be many section structures in the list which have the same name, since section structures from each file are independent from those for sections with the same from other files.

4.1.2 Address Allocation

Between Pass 1 and Pass 2, CLINK determines the in-memory address of each component of the output file. All sections are allocated in the order they are encountered. This order corresponds to the order of the section input list constructed during Pass 1.

Typically, each output section consists of a number of input sections. After allocation, the base and length of an output section represents the virtual address of the base of the entire section, and the length of the entire output section. The base and length of the input section describe just that portion of the output section comprised by that particular input section.

4.1.3 Pass 2

During Pass 2, CLINK creates the output file(s) by building the appropriate headers and copying parts of the input files to the output file(s) by traversing the input section list in the order constructed by Pass 1.

4.2 PROGRAM LOAD MODEL

FlexOS 386 supports a program load model called "fast load", which contains only CODE, DATA, and BSS sections. FlexOS 386 does not perform any relocation at load time so the .386 file must be statically located using specific rules. Figure 4-1 shows the address space of a .386 program at load time.

The fast-load algorithm used by the FlexOS 386 program loader consists of the following steps:

1. Read COFF file header; check that:
    a. Magic number is 514_O
    b. (Flags &= (F_EXEC & F_I386)) = (F_EXEC & F_I386).
2. Read FlexOS header; check that:
    a. Magic number 700_O
    b. Text_start = 0x1000 (4kb)
    c. Entry >= text_start
    d. Data_start = text_start + (tsize + 0x3ff)/0x1000 + max(0x1000, (stksize + 0x3ff)/0x1000). (The data must begin on the 1st 4Kb boundary after the end of the code and stack areas are each rounded up to 4Kb boundaries.
3. Allocate (tsize + 0x3ff)/0x1000 for code
4. Read CODE
5. Allocate max(0x1000, (stksize + 0x3ff)/0x1000) for stack
6. Allocate dsize + bsize for data and bss
7. Read DATA
8. Zero-fill BSS section
9. Begin program at "entry"

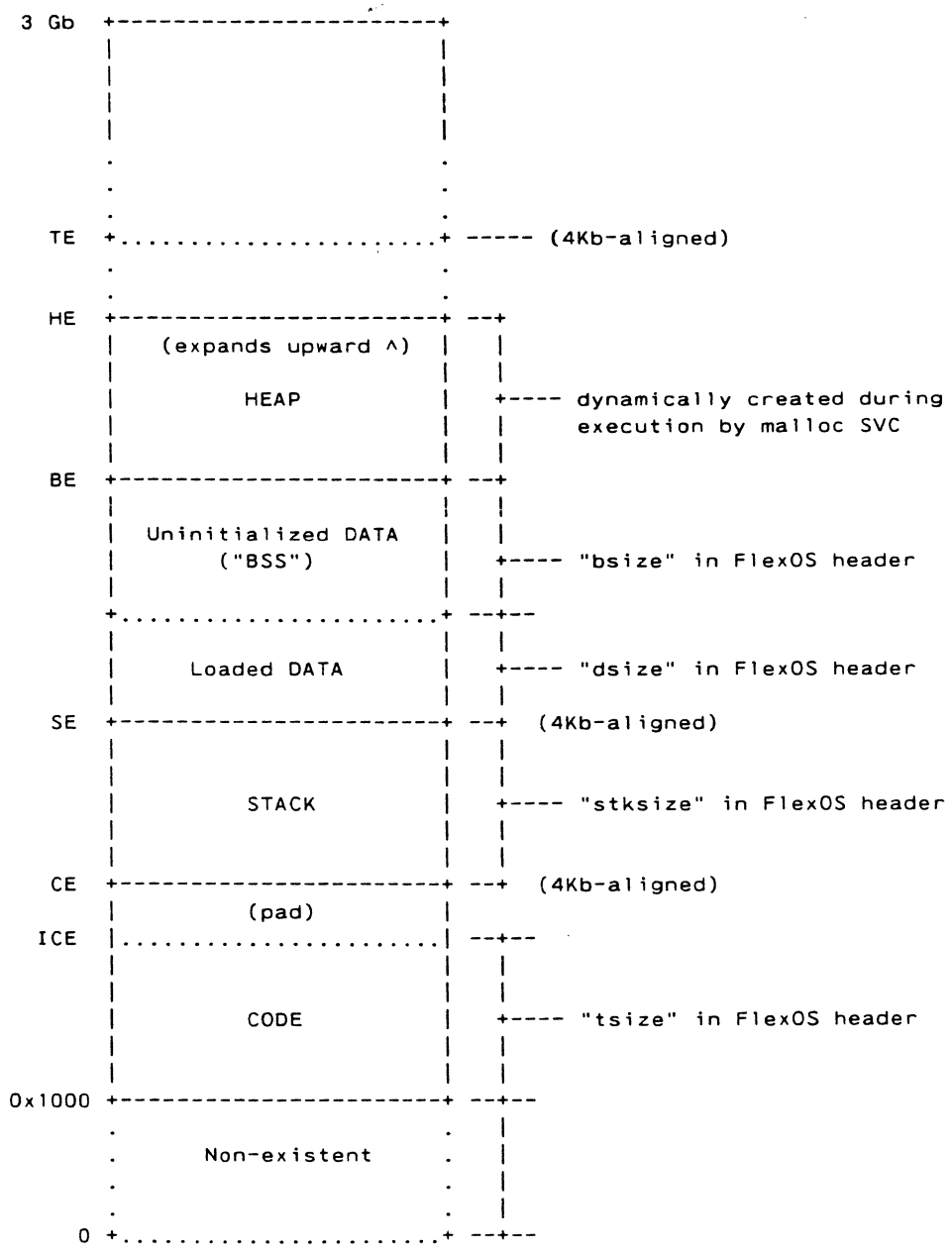

Figure 4-1. LOAD-TIME MEMORY MAP

The address symbols in the Figure 4-1 have the following definitions:

ICE (Initialized Code End)
: the address of the first byte after the loaded ".text" section of the .386 file.

CE (Code End)
: the address of the first byte after rounding the size of initialized code up to a 4Kb boundary. (CE = 0x2000 for applications with less than 4Kb of code in the ".text" section)

SE (Stack End)
: the address of the first byte after the required stack size specified in the optional header of the .386 file. (ESP is initialized to SE on 386 program load.)

BE (BSS End)
: the address of the first byte after the ".data" and ".bss" sections have been loaded from the .386 file. (Until allocation calls are made by the application, attempts to access this address or above generate an exception, and FlexOS 386 terminates the program.)

HE (Heap End)
: the address of the first byte after any dynamically allocated memory received from allocation SVCs.

TE (Trap End)
: (HE+0xFFF)/0x1000, which is the address of HE rounded up to 4Kb alignment. Attempting to access this address (or above) causes a trap to the OS.

Note: The following relation holds for .386 programs under FlexOS 386:

0x1000 < ICE <= CE < SE <= BE <= HE

4.3 USING CLINK

CLINK is invoked with a command of the form:

CLINK [filename][-option][filename] ...

where **filename** is either an object file or an input file.

Object files are either .O files from the assembler or compiler, or .LIB files from the librarian. If you enter a filename without a filetype or a period delimiter, CLINK first attempts to open the file using the name as entered. If the open fails, CLINK appends a **.o** to the name and attempts another open. If this open fails, CLINK displays a "file not found" message and terminates.

Input files contain CLINK command lines. That is, they are handled just as the command line itself, and may contain file names (which may be other input files) and linker options.

Linker options, are signified with a hyphen in the command line. Table 4-1 lists the command-line options.

Table 4-1. CLINK Options

Option	Purpose
cnum	Specify origin of .text section.
dnum	Specify origin of .data section.
ename	Mark "name" as the program entry point.
lname	Search for the library, LIBname.A in /lib, then /usr/lib.
Lpath	Specify a search path for libraries.
m	Produce a linkage map file.
oname	Specify output file name.
r	Retain relocation information in output file.
s	Strip symbols and line number data from output file.
Snum	Specify stack size.
T	Display execution times.
uname	Mark "name" as a undefined symbol.
v	Use Verbose mode.
V	Display linker version.
VSnum	Enter version number into FlexOS 386 header.

4.3.1 The cnum option

This option overrides the default relocated address of the .text section. The default address is 1000H; the FlexOS program loader expects this address. **num** must be a hexadecimal number.

4.3.2 The Dname option

This option overrides the default relocated addess of the .data section. The default is the next page-aligned address above the code and stack.

4.3.3 The ename option

This option causes CLINK to mark the address represented by the symbol **name** as the entry point of the program. FlexOS 386 begins execution at this point.

4.3.4 The lname option

This option causes CLINK to search a library named **lib**name.a, where name can be a string of up to 5 characters in length. A library is searched when its name is encountered, so the placement of a **-l** in the command line is significant. By default, libraries are assumed to be located in **/lib** and **/usr/lib**.

4.3.5 The Ldirectory option

This option inserts the specified directory in the chain of directories to be searched for libraries before **/lib** and **/usr/lib**. You must specify the **-L** option before the **-l** option if you want CLINK to search for the specified library in the alternate directory.

4.3.6 The m option

This option causes CLINK to produce a linkage map consisting of two parts:

- A **section table**, which displays input and output sections by name plus base addresses and lengths.
- A **symbol table**, which shows symbols and addresses corresponding to those symbols.

The map file name is the output file name with a file type of **.MAP**.

4.3.7 The oname option

This option causes CLINK to specify **name** as the name of the output file. By default, the output file name consists of the **first** filename encountered in the command line, concatenated with the filetype **.386**. If you do not specify an object file but simply link libraries, the output file name is the name of the first library encountered, with a filetype of .386.

4.3.8 The r option

This option causes CLINK to retain relocation information in the output file. Normally, the COFF object files produced by assemblers and compilers contain relocation entries which allow the modules to be combined with others and relocated by a linkage editor. FlexOS 386 does no relocation when loading application programs, so by default CLINK does not retain relocation information in the output file. To subsequently link the output file with others, this relocation information should be retained by using this option. Also, FlexOS 386 native-mode drivers must contain relocation information, requiring use of this option.

4.3.9 The s option

This option causes CLINK to remove symbols and line number entries from the retained and/or inserted into the output file and are useful for debugging.

4.3.10 The Snum option

This option changes the default stack size which CLINK inserts between .text and .data sections in the output file. The default stack size is 4 Kb. Although CLINK accepts any number entered using this option, FlexOS 386 rounds all stack sizes to the next higher multiple of 4Kb, and the default model loader expects to see the statically located .data section addressed at the top of that stack. For this reason, stack sizes which are multiples of 4Kb are normally used with this option.

4.3.11 The T option

This option causes CLINK tp display execution times.

4.3.12 The uname option

This option causes name to be entered as an undefined symbol in CLINK's symbol table. This is (perhaps) useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

4.3.13 The v option

This option causes CLINK to run in "verbose" mode, printing each module name.

4.3.14 The V option

This option causes CLINK to display its version number before processing the input files.

4.3.15 The (VSnum) option

This option instructs CLINK to insert **num** into the version field of the FlexOS 386 header in the output file. **num** must be a hexadecimal number.

4.4 CLINK ERROR MESSAGES

Table 4-2 lists the error messages that CLINK can display during processing.

Table 4-2. CLINK Error Messages

Error Message	Cause
file not found	CLINK could not locate an object file entered on the command line or within an input file.
file not closeable	FlexOS 386 returned an error when CLINK tried to close an object, input or output file.
file filename must be .o, .a or input file	CLINK could not identify filename as an object file, library file, or input file.
premature end of data in module mod-name, section sec-name	The object or library file is built incorrectly and could not be read by CLINK.
duplicate index-number in module mod-name, section sec-name	The object or library file is built incorrectly and could not be read by CLINK.
duplicate indices for symbol sym-name in module mod-name	The object or library file is built incorrectly and could not be read by CLINK.
types do not match for symbol sym-name in module mod-name	CLINK detected conflicting types for a symbol referenced in two or more modules.
auxents do not match for symbol sym-name in module mod-name	CLINK detected conflicting auxiliary entries for symbols referenced in two or more modules.
symbol sym-name undefined, first referenced in module mod-name	A symbol has been externally declared in one or more modules, but not defined in any module.
absolute symbol sym-mame assigned different value in module mod-name	An absolutely defined symbol cannot be re-defined or given different attributes.

Table 4-2. (Continued)

Error Message	Cause
absolute symbol sym-name defined as normal symbol in module mod-name	An absolutely defined symbol cannot be re-defined or given different attributes.
symbol sym-name redefined as absolute in module mod-name	An absolutely defined symbol cannot be re-defined or given different attributes.
duplicate definition of symbol sym-name in module mod-name	All symbol names must be unique.
unsupported storage class for symbol sym-name in module mod-name	The object or library file is built incorrectly and could not be read by CLINK.
symbol number sym-num not in index in module mod-name, section sec-name	The object or library file is built incorrectly and could not be read by CLINK.
special symbol sym-name cannot be defined in module mod-name	CLINK places certain special symbols in the output file. If the symbol name has been previously defined in your source code, CLINK outputs this error. Check the source code and re-define the symbol.
unordered reloc address addr in module mod-name, section sec-name	To improve performance, CLINK requires relocation entries in its object and library files exist in order of increasing virtual addresses. CASM and the Metaware High C compiler produce relocation data in this manner.
byte reloc for addr addr out of range in module mod-name, section sec-name	A relocatable byte reference to a symbol has resolved to a number which cannot be stored in one byte. Changing the link order may cause CLINK to position the target symbol closer to the reference.
word reloc for addr addr out of range in module mod-name, section sec-name	A relocatable word reference to a symbol has resolved to a number which cannot be stored in one word. Changing the link order may cause CLINK to position the target symbol closer to the reference.

Table 4-2. (Continued)

Error Message	Cause
unsupported reloc type for addr addr in module mod-name, section sec-name	CLINK supports the relocation types listed above in Section A.4. Relocation types used on other machines are not supported.
ifile filename nested too deeply at level number	CLINK does not allow input files to be nested more than 10 levels deep.
ifile filename contains nongraphic char, rest of file ignored	CLINK has encountered a non-printable character in an input file.
option optletter requires a name	Some command-line options require a parameter following the hyphen and option letter(s).
options r and s are incompatible	Relocation information requires the presence of symbol information. Retaining relocation entries with -r and stripping symbols with -s are incompatible operations.
unsupported option xxx ignored	The option letters displayed are not supported or recognized by CLINK.
decimal version number specified incorrectly	The version number supplied with the -VS option was entered incorrectly.
invalid library name	The library name supplied with the -L option was entered incorrectly.
invalid library directory name	The library directory name supplied with the -L option was entered incorrectly.

Table 4-2. (Continued)

Error Message	Cause
library name lib-name longer than 5 chars	The library file sub-name entered with the **-l** option may be a maximum of five characters, to fit within the library filename, LIBxxxxx.A.
library lib-name cannot be found	CLINK could not locate a library file entered on the command line or within an input file.

End of Section 4

CLIB - COFF LIBRARIAN

5.1 Introduction

CLIB™ is a utility program that combines multiple COFF format object files into an indexed library file. Such a library file can be processed by the linkage editor, CLINK to select those specific modules required to resolve external symbols of the program being linked.

A library file (filetype .LIB) includes complete copies of the object files, and extra data structures which CLINK uses to locate the modules containing the symbols for which it is searching. Each library contains a **symbol hash table**, which stores codes for all public symbols of the library modules. When searching the library to resolve an undefined symbol, CLINK performs the following actions:

1. Generate a hash code for the symbol.
2. Retrieve a pointer in the symbol hash table at the entry corresponding to that code.
3. Examine the information at that pointer to determine if the symbol is present in the library, and, if so, in which module of the library the symbol is defined.
4. Extract that module.

5.2 Command Line

CLIB is invoked by using a command line of the form:

CLIB libfilename [-option] [objfilename] ...

libfilename is the name of the library file you want to create or access. It can be any valid filename of 1 to 8 characters. **-option** is an option that controls CLIB operation as described below. **objfilename** is the name of a COFF object file (filetype .O). You can specify multiple object files with most options. Both the library and object files can have a path specification.

Table 5-0 lists the CLIB options, which follow a hyphen in the command line.

Table 5-1. CLIB Command Line Options

Option	Description
a	Add a module to a library
c	Print symbols by columns
d	Delete a module from a library
g	Print the Symbol table contents
h	Initiate the Help facility
Hnum	Use num for Hash Table size
m	Print the list of modules in a library
p	Print the list of modules by position
q	Quiet Mode
R	Build modules referenced list
r	Replace module in a library
ssymbol	Add/Delete a symbol from the symbol table
t	Print both the symbol table and list of modules
v	Verbose Mode
x	Extract a module from a module
?	Initiate the Help facility

5.2.1 The -a option

The -a option adds a module or modules to the specified library, creating the library if it does not exist. It is also used to associate a symbol or symbols with a module in an existing library.

Examples:

A>clib -a library.lib module.o

A>clib -a library.lib module1.o module2.o module3.o

A>clib -a library.lib -sSYMBOL1 -sSYMBOL2 modulex.o

5.2.2 The -c option

The -c option, when used with the -g option (print global symbols), displays the public symbols in the library in column format.

Example:

A>**clib -gc library.lib**

5.2.3 The -d option

The -d option deletes the specified module or modules from the library.

Example:

A>**clib -d library.lib modulex.o moduley.o modulez.o**

5.2.4 The -g option

The -g option prints all the public (global) symbols in all the modules of the specified library.

5.2.5 The -h option

The -h option displays all CLIB options with a short descriptive phrase.

5.2.6 The -Hnum option

The -H option specifies (in decimal bytes) the size of the symbol Hash Table which is built into a new library. The Hash Table size controls the maximum number of modules/symbols which can exist in the library. The default is 521 bytes.

Example:

A>**clib -a -H1000 newlib.lib firstmod.o**

5.2.7 The -m option

The -m option prints an alphabetized list of all modules in the specified library.

Example:

A>**clib -m library.lib**

5.2.8 The -p option

The -p option, when used with the -m option (print module names), displays the library modules in the order in which they exist in the library.

Example:

A>**clib -mp library.lib**

5.2.9 The -q option

The -q option puts CLIB in quiet mode so that it does not output any informative messages.

5.2.10 The -r option

The -r option replaces a module in the library with a file of an identical name on disk.

Example:

A>**clib -r library.lib module.o**

5.2.11 The -R option

The -R option builds an internal modules-referenced list which improves CLINK performance at the expense of increasing library size.

5.2.12 The -ssymbol option

The -s option, when used with the -a option adds a symbol to the library and associates it with the module whose name follows.

Example:

A>**clib -a library.lib -sSYMBOLNAME module.o**

5.2.13 The -t option

The -t option prints all the public (global) symbols and all the modules in the specified library.

Example:

A>**clib -t library.lib**

5.2.14 The -v option

The -v option puts CLIB in verbose mode so that when used with other options, CLIB displays informative messages.

Example:

A>**clib -tv library.lib**

5.2.15 The -x option

The -x option extracts a module from the specified library, creating a file with the same name as the module name, but leaving the library unchanged.

Example:

```
A>clib -x library.lib mod1.o
```

5.2.16 The -? option

The -? option performs the same function as the -h option.

5.3 CLIB Error and Warning Messages

Table 5-2 lists the warning messsages CLIB can emit when processing files.

Table 5-2. CLIB Warning Messages

Message	Description
Too many arguments - ignored	The command line contained extra arguments which CLIB does not understand. The extra arguments have been ignored.
Cannot open module file - ignored	One module (file) in a list of modules could not be opened by CLIB. If the other modules in the list can be opened, they are processed by CLIB.
A Module By This Name Already in Library - ignored	When attempting to add a module to an existing library, CLIB detected the presence of a previous module of the same name. The command for the redundant operation has been ignored.
Module Not In The Library	CLIB could not find the specified module in the library.
A Symbol By This Name Already in Library - ignored	When a module is added which contains an external (public) symbol whose name duplicates a symbol already present in the library, CLIB issues this message and the reference to the new symbol is not entered into the library symbol table.

Table 5-3 lists the error messages that cause immediate termination of CLIB.

Table 5-3. CLIB Error Messages

Message	Description
Internal Error	CLIB issues this message when it detects the library or a module being processed is incorrectly built. Check the operation of the assembler or compiler which produced the module.
Cannot Open Library File	CLIB cannot open the library file specified in the command line.
Cannot Create Library File	When attempting to create a library file for the first time, CLIB received an error message from FlexOS. The reason may be one of several: media full, media write-protected, media directory full, etc.
Not a Library File	The library file specified on the command line was not recognized by CLIB as being in DRI library format.
Cannot Open Library File For Updating	The library file specified on the command line could not be opened for updating. This is normally the result of the file being opened exclusively by another user.
Library Built with Incompatible Version of CLIB	The library file specified on the command line was created by an earlier, incompatible version of CLIB.
No Module(s) to Work On	A library file and options were specified on the command line which require the specification of a module or modules.

End of Section 5

CSID - SYMBOLIC DEBUGGER

6.1 INTRODUCTION

CSID™ is a symbolic debugger designed to use with the FlexOS 386 operating system and the Intel® 80386 processor. CSID features:

- Symbolic assembly and disassembly
- Expressions involving hexadecimal, decimal, octal, binary, ASCII, and symbolic values
- Execution breakpoints with pass counts
- Data breakpoints
- Macro definitions for complex commands

6.2 TYPOGRAPHICAL CONVENTIONS

The following typographical conventions are employed to illustrate CSID's command and output structures:

- Commands appear in UPPERCASE characters and their arguments appear in lower case characters. This convention is used only to distinguish the command from its arguments. Typically, you enter all CSID command characters in lower case.
- In CSID command examples, user input is displayed in **bold print**.
- Some examples of CSID output use horizontal and/or vertical ellipses (.....) to illustrate the continuation of an output pattern.
- Ctrl indicates the CONTROL key on the keyboard.
- Curly braces {} are used to signify an optional parameter.
- A vertical bar | indicates a choice between the items it separates.

6.3 STARTING CSID

You start CSID by entering a command in the form:

CSID {filespec} {-options}

filespec is the name of the file to be debugged, including an optional pathname. If you do not specify a pathname, CSID uses the pathname currently specified in the pathname table. If you do not enter a filetype, CSID assumes a .386 filetype. **Options** is one or more of the options described below.

If you do not specify a **filespec**, CSID simply displays its sign-on banner and prompt character (#), and awaits commands.

CSID does not allow multiple executable files to be loaded and debugged simultaneously. When a program file is loaded for execution, it overwrites the previously loaded file.

6.4 CSID COMMAND-LINE OPTIONS

CSID command-line options are divided into two categories: **process control** options and **windowing** options. All options must be located at the end of the command line, with multiple options separated by white space (tabs or blank space).

6.4.1 Process Control Options

Table 6-1 lists the process control options.

Table 6-1. CSID Process Control Options

Option	Purpose
T args	T is followed by the arguments for the program being debugged. Use T with non-interactive programs where one or more arguments are required for the debugged program. If used, T must be the last option in the command line.
M value	M is followed by a hexadecimal value specifying the maximum memory size (in kilobytes) for the debug process. A large maximum memory size is recommended. The default is 6000H.
F filespec	F is followed by the name of a CSID command file to be read. A command file is a file containing a list of CSID commands, including macros.

6.4.2 Windowing Options

When using CSID to debug a program, the process executing the code under control of CSID is referred to as the debug process. CSID allows you to dedicate a section of your screen to the debug process. This section of the screen is referred to as the debug process window.

The windowing options determine the size of the debug process window when the debug process has control, as well as the size of the window when the debug process is complete and control returns to CSID. All of the window sizes are entered as hexadecimal values.

Table 6-2 lists the windowing options.

Table 6-2. CSID Windowing Options

Option	Purpose
R value	R is followed by the hexadecimal value specifying the maximum horizontal (row) size of the debug process window. The range is typically 0 - 19H, but the maximum size can vary between systems. The default value is the same size as CSID's original window size.
C value	C is followed by the hexadecimal value specifying the maximum vertical (column) size of the debug process window. The range is typically 0 - 50H, but the maximum size can vary between systems. The default value is the same size as CSID's original window size.
W value	W is followed by the hexadecimal value specifying the continuous vertical size of the debug process window. When you establish the window size using the R and C options, you can use W to specify how much of the window will remain showing at all times. You can change this value from the CSID command line with the SET command (see Section 9.2).

The following are example CSID command lines:

A>csid hello.386
Start CSID and load the command file "hello.386" as the debug process.

A>csid -fmacfile
Start CSID, and read the macros from the file, "macfile".

A>csid fgrep -m 80 -t "trail" *.a *.lib
Start CSID and load the command file "fgrep.386". Specify the maximum memory size for fgrep as 128 Kbytes. The command tail "trail" *.a *.lib are the arguments for fgrep, which specify fgrep to search for the string "trail" in the files *.a and *.lib.

A>csid fgrep -t "farewell" *.c

Start CSID and load the command file "fgrep.386". The command tail "farewell" *.c are given as the arguments.

A>csid -w 10 -r 0f -c 32

Start CSID, and set the debug process window size to 15 rows by 50 columns. The W option specifies that 16 rows of the debug process are displayed when CSID is invoked.

6.5 CSID COMMAND CONVENTIONS

The CSID command prompt is a pound sign, #. A valid CSID command can have up to 256 characters and must be terminated with a carriage return.

A CSID command can be followed by one or more arguments. The arguments can be symbolic expressions, filenames, or other information, depending on the command. Arguments are separated from each other by commas or spaces.

Most CSID commands require one or more addresses as operands. Enter an address as follows:

nnnnnnnn

where **nnnnnnnn** represents a 32-bit address.

CSID does not allow you to randomly access any area in memory. Access is limited to areas read into CSID using the READ command and areas of a debugged process read into CSID using the LOAD command, or by means of the invocation line. It is not possible to simultaneously have a debugged process (read in by the LOAD command or command line) and a file (read in by the READ command) resident in CSID.

6.6 LINE EDITING KEYS

CSID does not process the command line until you enter a carriage return. You can edit the command line using the line-editing keys:

Ctrl-A	Move backward to beginning of previous word.
Ctrl-D	Move forward one character.
Ctrl-E	Recall the previous line from the history buffer, moving upward in buffer from newest to oldest. The end of the buffer is an empty line.
Ctrl-F	Move forward to the beginning of the next word.
Ctrl-G	Delete the character under the cursor.
Ctrl-H	Delete the previous character (same as backspace).

Ctrl-J	Same as Ctrl-M, except the line is not saved.
Ctrl-K	Delete from the current cursor position forward to the end of the line.
Ctrl-M	End the current line and save the line in the history buffer if necessary. Can be used at any position on the line (same as enter.)
Ctrl-Q	Move to beginning of line.
Ctrl-R	Turn Search Mode ON/OFF for the current line. The Search Mode finds only those lines that match the character(s) to the left of the cursor. Use Ctrl-E/Ctrl-X to move through matching entries, or type more characters to match. After entering the line, the Search Mode reverts to the default set by Ctrl-_.
Ctrl-S	Move backward one character.
Ctrl-T	Delete forward to the beginning of the next word.
Ctrl-U	Delete backward to the beginning of the line.
Ctrl-V	Toggles between INSERT and OVERSTRIKE mode. The current mode is saved for future line editing (the default is initially INSERT).
Ctrl-W	Move to the end of the line.
Ctrl-X	Recall the next line from the history buffer, moving downward in the buffer from oldest to newest.
Ctrl-Y	Delete the entire line and save it if it is new (or edited).
Ctrl-\	Enter the next character without special interpretation (i.e. to enter a Ctrl-G, type Ctrl-\Ctrl-G.
Ctrl-_	Toggle the default Search Mode ON/OFF. Normally, the Search Mode is OFF for each new entry. Ctrl-_ sets the default mode to ON and also sets the current Search Mode to ON. At the start of the next line, the Search Mode is enabled unless turned off temporarily with Ctrl-R, or the default is set to OFF with another Ctrl-_.

6.7 CSID COMMAND SUMMARY

Table 6-3 summarizes the CSID commands. CSID commands are defined individually in Chapters 8 and 9.

Table 6-3. CSID Command Summary

Command	Action
ABORT	Abort the debug process
ASSEMBLE	Enter assembly language statements
ASSIGN	Assign function return value to a symbol
BREAK_IF	Enable conditional breakpoint
CALCULATE	Calculate the value of an expression
CLOSE	Close output redirection or user-input command file
COMPARE	Compare memory
DEFINE	Define a macro/symbol
DISABLE	Disable breakpoints
DISPLAY	Display memory, breakpoints, or options
ELSE	Specify ELSE command conditional clause
ENABLE	Enable breakpoints
ENDIF	Specify end of ENDIF conditional statement
EXIT	Exit CSID
FILL	Fill memory block with a constant value
GO	Begin execution of debug process
IF	Enable conditional execution of commands
LIST	List assembly code
LOAD	Load 386 program for debugging
MOVE	Move memory contents from one location to another
READ	Read macros, symbols, or data file
REMOVE	Remove breakpoints
SEARCH	Search for a value in memory
SET	Set memory, breakpoints, or options
TRACE	Trace through an instruction
WRITE	Write memory or macros to specified file
?	List CSID commands
HELP	List CSID commands with options

End of Section 6

CSID Expressions

7.1 Introduction

CSID expressions can use symbol names from the COFF command file, as well as literal values in binary, octal, hexadecimal, decimal, or ASCII character string form. You can combine these literal values with arithmetic operators to provide access to subscripted and indirectly-addressed data or program areas.

7.2 Literal Hexadecimal Numbers

CSID normally accepts and displays values in hexadecimal. A literal hexadecimal number in CSID consists of one or more contiguous hexadecimal digits. If the beginning digit is alphanumeric, it must be preceded with a zero. If you type eight digits, the leftmost digit is most significant and the rightmost digit is least significant. If the number contains more than eight digits, the rightmost eight are recognized as significant, and the remaining leftmost digits are discarded.

The following examples show the hexadecimal input value and the corresponding value stored by CSID.

Input Value	Hexadecimal
1	0001
100	0100
0fffe	FFFE
10000	10000
10000000	10000000
100000000	00000000
38001	38001
380010000	80001000

7.3 Literal Decimal Numbers

Enter decimal numbers with a trailing decimal point. The number must consist of one or more decimal digits (0 through 9), with the most significant digit on the left and the least significant digit on the right. Decimal values are padded or truncated according to the rules of hexadecimal numbers when converted to the equivalent hexadecimal value.

The following examples show the hexadecimal values produced by the input values.

Input Value	Hexadecimal Value
9.	0009
10.	000A
256.	0100
65535.	FFFF
65545.	10009

7.4 Literal Character Values

CSID accepts one to four ASCII characters enclosed in apostrophes (single quote marks) as literal values in expressions. The leftmost character is the most significant, and the rightmost character is the least significant. Strings having more than four characters are not allowed in expressions, except in the SEARCH command, as described in Section 9.4.

Note that the enclosing apostrophes are not included in the character string, nor are they included in the character count. The only exception is when a pair of contiguous apostrophes is reduced to a single apostrophe and included in the string as a normal graphic character (see examples below).

The following examples, show the hexadecimal values produced by the input strings. Note that uppercase ASCII alphabetics begin at the encoded hexadecimal value 41; lowercase alphabetics begin at 61; a space is hexadecimal 20 and an apostrophe is hexadecimal 27.

Input String	Hexadecimal Value
'A'	41
'AB'	4142
'aA'	6141
''''	0027
''''''	2727
' A'	2041
'A '	4120

7.5 Register Values

You can use the contents of a register by specifying the register name wherever a number is valid. Table 9-2 lists the 80386 register names.

7.6 Symbol References

There are two basic ways to reference values associated with symbols:

s
[s]

The form **s** gives the 32-bit value associated with the symbol **s** in the COFF symbol table. The form **[s]** gives the 32-bit value pointed to by s.

The following example illustrates these forms. Given the memory values and symbols defined below:

Memory location	COFF symbol	Memory value
00000100	Gamma	02
00000101		3E
00000102	Delta	4D
00000103		22

then the symbol references shown below on the left gives the hexadecimal values shown on the right. Recall that multi-byte memory values are stored with the least significant byte first. Therefore, the word values at 0100 and 0102 are 3E02 and 224D, respectively.

Gamma	00000100
delta	00000102
word[gamma]	3E02
word[delta]	224D
byte[gamma]	02
byte[delta]	4D
[gamma]	3E02

7.7 Qualified Symbols

Duplicate symbols can occur in the symbol table due to separately assembled or compiled modules that independently use the same name for different subroutines or data areas. Block structured languages allow nested name definitions that are identical, but nonconflicting. Thus, CSID allows reference to "qualified symbols" that take the form

S1\S2\ . . . \Sn

where **S1** through **Sn** represent symbols present in the table during a particular session.

CSID always searches the Symbol table from the first to last symbol in the order the symbols appear. For a qualified symbol, CSID begins by matching the first **S1** symbol, then searches for a match with symbol **S2**, continuing until symbol **Sn** is matched. If this search and match procedure is not successful, CSID prints an error message. Suppose that part of the Symbol table appears as follows:

```
00000100 A  00000300 B  00000200 A  00003E00 C  000020F0 A  00000102 A
```

Then the unqualified and qualified symbol references shown below on the left produce the hexadecimal values shown on the right.

Symbol Reference	Hexadecimal Value
A	00000100
WORD[A]	2D04
A/A	00000200
C/A/A	0000:0102
BYTE[C/A/A]	5E
B/A/A	000020F0

7.8 Expression Operators

Literal numbers, strings, and symbol references can be combined into symbolic expressions using any of the following operators:

Table 7-1. CSID Expression Operators

Operator	Description
+	addition or unary positive
-	subtraction or unary negative
*	multiplication
/	unsigned division
MOD	return remainder of division
AND	logical AND
NOT	logical NOT
OR	logical OR
XOR	logical eXclusive OR
SHR	shift right
SHL	shift left
EQ	Equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
NE	Not Equal to
LOW	return low-order byte of 16-bit or greater sized operand
LOWW	return low-order half-word of 32-bit or greater sized operand
HIGH	return high-order byte of 16-bit or greater sized operand
HIGHW	return high-order half-word of 32-bit or greater sized operand

CSID evaluates the expression from left to right, producing a 32-bit address at each step. Overflow and underflow are ignored as the evaluation proceeds. The final value becomes the command parameter, whose interpretation depends upon the particular command preceding it.

In commands that specify a range of addresses, the ending address can be indicated as an **count** from the starting address by preceding the desired count with a plus sign. For example, the command

```
# dm 0fd0,+512
```

displays the memory from offset address FD00 to FF00. CSID does not allow use of the unary plus operator at other times.

7.9 Sample Symbolic Expressions

To be supplied

End of Section 7

BASIC CSID COMMANDS

This chapter describes the CSID commands that you use most often to debug programs. Additional commands, including those that use CSID's more advanced features are described in Chapter 9.

8.1 LOAD COMMAND

The LOAD command loads a .386 file into memory for subsequent execution. The form is:

LOAD {filespec} {-options}

where **filespec** is the name of the .386 executable file to load. If you do not specify a filetype, CSID assumes a .386 filetype. CSID loads any symbols from the COFF file into its own memory. CSID issues an error messsage if a file does not exist or cannot be successfully loaded in the available memory space. **options** are the same options described in Section 6.4.

When the file is loaded, CSID displays the start and end addresses of the CODE and DATA sections[2]. After you execute the program, you can redisplay this information using DISPLAY LOAD_INFO (see Section 9.1), and if the program has created a heap, CSID displays the start and end addresses of the HEAP section. The LOAD command releases memory allocated by any previously loaded programs. Therefore, only one file at a time can be loaded for execution.

Example:

#load test Load file TEST.386 from default directory

8.2 READ COMMAND

The READ command has three forms:

(1) READ FILE filespec
(2) READ COMMANDS filespec
(3) READ SYMBOLS filespec

With all three forms, **filespec** is the name of the file you want to read. CSID issues an error message if the file does not exist or there is not enough memory to load the file.

Form 1 reads a file into memory, computes, allocates, and displays the starting and ending addresses of the memory block occupied by the file.

---

[2]DATA contains both the COFF .data and .bss sections

You can redisplay this information later with the DISPLAY LOAD_INFO command. Form 1 allows you to display, modify, and then write out the file that is read in.

Note: Form 1 does not free any memory allocated by a previous READ. Therefore, you can read a number of files into memory and concatenate them together in the order in which they are read in.

With Form 1, **filespec** can be any file. If you use READ FILE after a process is already loaded with LOAD (or from the command line), CSID stops the debug process.

Form 2 reads in and executes previously defined CSID commands. **filespec** must be an ASCII file containing CSID commands. Form 2 is generally used to read predefined macros, macros saved with the WRITE MACROS command, or commands written with the WRITE SESSION command (see Section 9.10).

Form 3 reads in symbols from any file. **filespec** should be a COFF file (either .386 or .O).

Examples:

#read file banner.386
Read file BANNER.386 into memory.

#read symbols test
Read symbols from executable file TEST into memory.

8.3 EXIT/ABORT COMMANDS

The EXIT command terminates CSID if no process is being debugged. If a debug process is running, use ABORT to stop the process.

The forms are:

EXIT

ABORT

Note: CSID does not automatically save files. If you modify a file and wish to save it, you must write the modified file to disk using the WRITE command described in Section 9.10 before exiting CSID.

8.4 DISPLAY MEMORY COMMAND

The DISPLAY MEMORY command displays the contents of memory as either 8-bit, 16-bit, or 32-bit hexadecimal values and corresponding printable ASCII characters. The form is:

DISPLAY MEMORY {s,e}

s is the starting address, and **e** is the ending address. If you do not specify a starting address, the default is the end of the last display.

After a LOAD command, the default is the first byte of the DATA section. After a READ FILE command, the default is the first byte of the file read in. The last address displayed becomes the default starting address for the next display. The display stops at the end of memory, regardless of the ending address specified.

Memory is displayed on one or more lines, with each line showing values of up to 16 memory locations in the form:

```
00000000 bb bb . . . bb a . . . a

00000000 wwww wwww . . . wwww a . . . a

00000000 llllllll llllllll . . . llllllll a . . . a
```

where **00000000** is address, **bb**'s represent the 8-bit contents of the memory location, the **wwww**'s 16-bit contents, and the **llllllll**'s 32-bit bit contents in hexadecimal. The **a**'s represent the contents of memory in ASCII. A period represents any nongraphic ASCII character.

During a long display, you can control scrolling with Ctrl-S/Ctrl-Q, or stop the display by typing any character at the console.

Examples:

#db 0f00,0f23	Display memory bytes from offset F00H through F23H.
#db array+=i,+10	Display 10 bytes starting at location ARRAY (i).
#dw esp	Display the value at the top of stack in word format.
#dm esp	Display the value at the top of stack in byte format.
#dw 80,0ff	Display memory words from offset 80H through FFH.

8.5 LIST COMMAND

The LIST command lists the contents of memory in disassembled CASM statements. The form is:

```
LIST {s,e}
```

where **s** is the starting address, and **e** is the ending address. If e is a count, then it specifies the number of lines to disassemble. If you do not specify a starting address, the default is the end of the last list. After a LOAD command, the default address is the EIP. After a READ FILE command, the default address is the first byte of the file read in. The last address displayed becomes the default starting address for the next list unless another starting address is specified (see SET LIST_START in Section 9.2). The list stops at the end of memory, regardless of the ending address specified.

Each disassembled instruction takes the form:

label:
00000000 prefixes opcode operands memory value

where **label** is the symbol whose value is equal to the address **00000000**, if such a symbol exists; **prefixes** are LOCK, REPEAT, ASP, and OSP prefixes; **opcode** is the 386 mnemonic for the instruction. **operands** is a field containing 0, 1, or 2 operands, as required by the instruction. If the instruction references a memory location, the LIST command displays the contents of the location in the **memory value** field as a BYTE, WORD, or LONG as indicated by the instruction.

If the memory location being disassembled is not a valid 80386 instruction, CSID displays

??= nn

where **nn** is the hexadecimal value of the contents of the memory location.

By default, the LIST command lists 12 disassembled instructions from the current list address. You can change the number of lines listed with the SET LIST_LENGTH command (see Section 9.2). During a long list, you can control scrolling with Ctrl-S/Ctrl-Q, or stop the display by typing any character at the console.

Examples:

#list	Disassemble 12 instructions from the current default list address.
#list 243c,244e	Disassemble instructions from 243CH through 244EH.
#list find,+20	Disassemble 20H lines from the label FIND.
#list err+3	Disassemble 12 lines of code from the label ERR plus 3.
#list err,err1	Disassemble from label err to label err1.

8.6 GO COMMAND

The GO command transfers control to the debug process and optionally sets one or two temporary breakpoints. A **temporary** breakpoint is a breakpoint that causes execution to stop immediately preceding an instruction, but is removed after it has been encountered, or a new temporary breakpoint is entered.

The form is:

GO {s,b1,b2}

where **s** is the address where program execution is to start, and **b1** and **b2** are addresses of breakpoints. If you do not specify a starting address, the default is the current value of the EIP.

When the debug process receives control, it executes in real time until a breakpoint is encountered. CSID then regains control, displays the current CPU state and the next instruction to execute.

Examples:

#go	Begin program execution at address given by the EIP register with no temporary breakpoints set.
#go start,error	Begin program execution at label START, setting a temporary breakpoint at label ERROR.
#go error,[esp]	Continue program execution at address given by the EIP register, with temporary breakpoints at label ERROR and at the address at the top of the stack.

8.7 TRACE COMMAND

The TRACE command traces program execution. The forms are:

(1) TRACE SINGLE {count,m1,m2}
(2) TRACE WHOLE {count,m1,m2}

The **count** is a number ranging ranging from 1 to 0FFFFFFFH indicating how many instructions you want to execute. Each time an instruction executes, the count decrements by 1. If you do not specify a count, the default is 1.

m1 is the name of a macro you want to run at every trace. **m2** the name of a macro you want to run when the count reaches zero. See Section 9.12 for a complete discussion of macros.

Form 1 traces a single instruction, with the following exceptions:

- When tracing an SVC call, the entire call is treated as one program step and executed in real time.
- When tracing a MOV or POP whose destination is a segment register, the CPU executes the next instruction immediately.

Form 2 traces execution without breaking for calls to subroutines. If the traced instruction is a CALL or similar instruction[3] CSID sets a temporary TRACE WHOLE breakpoint immediately following the instruction. The program then executes in real time until the breakpoint is encountered. This allows tracing at a high level of the program, ignoring subroutines already debugged.

---

[3]CALLF,CMPSx,LODSx, MOVSx,SCASx,STOSx,OUTSx,or INSx. x is either B, W, or L.

After each program step is executed, CSID displays the current CPU state, the next instruction to be executed, the symbolic name of the instruction operand (if any), and the contents of the memory location(s) referenced by the instruction (if appropriate).

If a symbol has a value equal to the instruction pointer (EIP), the symbol name followed by a colon is displayed on the line preceding the CPU state display.

With both forms, control transfers to the program under test at the address indicated by the EIP register. If you do not specify the number of program steps, TRACE executes one program step. Otherwise, CSID executes the number of program steps specified by **count** and displays the CPU state before each step. You can stop a long trace by typing any character at the console.

Examples:

#t	Trace one program step.
#t 0ffff	Trace 0FFFFH (65535) steps.

8.8 BREAKPOINT COMMANDS

CSID allows you to set, display, enable, disable, and remove permanent breakpoints. A **permanent** breakpoint is a breakpoint that causes execution to stop immediately preceding an instruction, and remains in effect until you explicitly remove or disable it. CSID can set up to 13 permanent breakpoints at a time. CSID also supports **temporary** breakpoints to be set with the GO command (see Section 8.6).

CSID also supports two types of **data** breakpoints. An ACCESS data breakpoint causes execution to stop immediately after data has been accessed (Read or Write). A MODIFY data breakpoint causes execution to stop immediately after data has been modified (Write).

The CSID commands to manipulate breakpoints are:

(1) SET EXECUTE_BP address {count,m1,m2}
(2) SET ACCESS_BP|MODIFY_BP length address {count,m1,m2}
(3) DISPLAY bptype {address}
(4) DISABLE bptype {address}
(5) ENABLE bptype {address}
(6) REMOVE bptype {address}

Form 1 sets (creates and enables) a breakpoint at the specified **address**. The **count** is a number ranging ranging from 1 to 0FFFFFFFH indicating how many times you want the instruction at the breakpoint to execute. Each time the breakpoint is encountered, the count decrements by 1. If you do not specify a count, the default is 1. If a breakpoint is already active at the given address, the count is changed to **count**.

m1 is the name of a macro you want to run at every occurance of the breakpoint.

m2 the name of a macro you want to run when the count reaches zero. (See Section 9.12 for a complete discussion of macros.)

Form 2 sets data breakpoints (either ACCESS or MODIFY), and is similar to form 1. The address should be aligned to the specified **length**, whether BYTE, WORD, or LONG.

Note: The 80386 processor has four hardware breakpoint registers that CSID uses for breakpoint support. If the address is not aligned, CSID automatically sets multiple breakpoints to simulate the proper alignment, thereby making fewer available to be set by the user. For example, if a WORD length breakpoint is set on a BYTE boundary, CSID sets two BYTE-length breakpoints to simulate the single WORD-length breakpoint.

Form 3 displays an active breakpoint at the specified address. The **bptype** is one of the following:

ACCESS_BP
MODIFY_BP
EXECUTE_BP

If you specify a type but no address, CSID displays all the breakpoints with the given type. If you specify BREAKPOINTS, CSID displays **all** breakpoints of all types. The display has the form:

type 00000000 n m1 m2 disabled/enabled *temporary/*trace whole

where **type** is one of the following:

ACCESS length
MODIFY length
EXECUTION OF

00000000 is the address, **n** is the count, **m1 m2** are macros. The display also indicates whether the breakpoint is disabled or enabled, and if the breakpoint is only temporary (see the GO command), or it is internally enabled by the TRACE WHOLE command.

Form 4 disables but do not remove the breakpoint at the specified address. If you specify a type but no address, CSID displays all the breakpoints with the given type. If you specify BREAKPOINTS, CSID displays **all** breakpoints of all types.

Form 5 re-enables a breakpoint at the specified address that was previously disabled by the DISABLE command. If you specify a type but no address, CSID displays all the breakpoints with the given type. If you specify BREAKPOINTS, CSID displays **all** breakpoints of all types.

Form 6 removes the breakpoint at the specified address. If you specify a type but no address, CSID displays all the breakpoints with the given type. If you specify BREAKPOINTS, CSID displays **all** breakpoints of all types.

Examples:

#display breakpoints
Display active permanent breakpoints.

#set execution error
Set permanent breakpoint at label ERROR.

#set execution print,17
Set permanent breakpoint at label PRINT with count of 17H.

#remove breakpoints
Clear all permanent breakpoints.

#remove breakpoints error
Clear permanent breakpoint at label ERROR.

End of Section 8

ADDITIONAL CSID COMMANDS

This section describes additional CSID commands that can be useful when debugging programs.

9.1 DISPLAYING OTHER INFORMATION

In addition to displaying memory, the DISPLAY command has other forms that can display various types of information. The forms are:

(1) DISPLAY BASE
(2) DISPLAY MEMORY_LENGTH
(3) DISPLAY MEMORY_START
(4) DISPLAY ECHO
(5) DISPLAY FLAGS flag_list
(6) DISPLAY LIST_LENGTH
(7) DISPLAY LIST_START
(8) DISPLAY LOAD_INFO
(9) DISPLAY REGISTERS register_list
(10) DISPLAY SYMBOL symbol_name
(11) DISPLAY USE_MODE
(12) DISPLAY WINDOW LOCATION
(13) DISPLAY WINDOW SIZE

Form 1 displays the default base (radix) of CSID. The display has the form:

Current base is b

where **b** is one of the following:

- 2 - base 2 (binary)
- 8 - base 8 (octal)
- 10 - base 10 (decimal)
- 16 - base 16 (hexadecimal)

Form 2 displays the currently defined length (number of bytes) for the DISPLAY MEMORY command. The display has the form:

Current length is n

Form 3 displays the currently defined default starting address for the DISPLAY MEMORY command. The display has the form:

DISPLAY starting address is 00000000

where **00000000** is the 32-bit address.

Form 4 displays the currently defined default setting for echoing CSID output, whether ON or OFF.

Form 5 displays the current state of the CPU flags. The display has the form:

```
xxxxxxxxx
```

where **x** represents either a hyphen, indicating the corresponding flag is not set (0), or a single-character abbreviation of the flag name, indicating the flag is set (1). Table 9-1 lists the abbreviations of the flag names.

Table 9-1. Flag Name Abbreviations

Character	Name
O	Overflow
D	Direction
I	Interrupt Enable
T	Trap
S	Sign
Z	Zero
A	Auxiliary Carry
P	Parity
C	Carry

Form 6 displays the currently defined length (number of lines) for the LIST command. The display has the form:

```
Current length is n
```

Form 7 displays the currently defined starting address for the LIST command. The display has the form:

```
LIST starting address is 00000000
```

where **00000000** is the 32-bit address.

Form 8 displays information about the file loaded with the LOAD or READ commands. If you load the file with READ, form 8 displays the starting and ending addresses of the memory block where the file is loaded. If you load the file with LOAD, form 8 displays the starting address and length in bytes for the CODE, DATA, and HEAP sections of the COFF file. The display has the form:

```
.          address     length
code       00000000    00000000
data       00000000    00000000
heap       00000000    00000000
```

where **00000000** is the 32-bit address.

Form 9 displays the contents of the specified **register_list**, which contains the names of 80386 CPU registers (see Table 9-2). If you do not specify which registers, the default is the macro **'allregs'** (see Section 9.12).

The display has the form:

```
EAX   EBX   ECX   . . .   SS    ES    IP
xxxx  xxxx  xxxx  . . .   xxxx  xxxx  xxxx
```

Form 10 displays the address of the specified **symbol_name**. FlexOS wildcards are allowed for the symbol name. The display has the form:

```
00000000  symbolname
```

where **00000000** is the 32-bit address.

Form 11 displays the current USE mode of CASM (see Section 3.2.1, either 16-bit or 32-bit. The display has the form:

```
Current Use Mode is U
```

where **U** is 16 or 32.

Form 12 displays the current location (row and column) of the debug process's window. The display has the form:

```
Current window size is r,c
```

where **r** is the row value and **c** is the column value.

Form 13 displays the current size (number of rows) of the debug process's window. The display has the form:

```
Current window location is r,c
```

where **r** is the row value and **c** is the column value.

Examples:

TBS

9.2 SET COMMAND

The SET command changes the contents of memory and sets other values. The forms are:

(1) SET BASE base
(2) SET MEMORY_LENGTH length
(3) SET MEMORY_START address
(4) SET FLAGS flag_list
(5) SET LIST_LENGTH length
(6) SET LIST_START address
(7) SET MEMORY address {,value}
(8) SET ECHO
(9) SET REGISTERS register_list
(10) SET USE_MODE mode
(11) SET WINDOW LOCATION row,column
(12) SET WINDOW SIZE row

Form 1 sets the default base (radix) for CSID. The base is one of the following:

- 2 - base 2 (binary)
- 8 - base 8 (octal)
- 10 - base 10 (decimal)
- 16 - base 16 (hexadecimal)

Form 2 sets the default length (number of bytes) for the DISPLAY MEMORY command.

Form 3 sets the default starting address of the DISPLAY MEMORY command.

Form 4 sets the CPU flags specified in **flag_list** as follows:

'f1 f2 f3 ... f9'

where **f1 f2** ... are the single letter abreviations of CPU flags listed in Table 9-1. CSID responds by displaying the name of the flag followed by its current state. If you enter a carriage return, the flag's state does not change. If you enter a valid value (either 0 or 1), the flag's state changes to that value.

Form 5 sets the default length (number of lines) for the LIST command.

Form 6 sets the default starting address for the LIST command.

Form 7 sets memory at the specified **address**, with an optional **value**. If you do not specify a value, CSID prompts for a value. CSID displays the memory address and its current contents on the following line. The display has one of the forms:

```
00000000 bb
00000000 wwww
00000000 llllllll
```

where 00000000 is the address and **bb** is the contents of memory in BYTE format, **wwww** in WORD format, and **llllllll** in LONG format, depending on which form of the command you use.

You can choose to alter the memory location or to leave it unchanged. If you enter a valid expression, CSID replaces the contents of memory with the value of the expression. If you do not enter a value, CSID does not replace the contents of memory but displays the next address. In either case, CSID continues to display successive memory addresses and values until you enter a period on a line by itself, or until CSID detects an invalid expression.

You can enter a string of ASCII characters, delimited with apostrophes (single qubtation marks). The characters between the quotation marks are placed in memory starting at the address displayed. The next address displayed is the address following the character string.

Form 8 sets CSID's output mode either ON or OFF.

Form 9 sets the registers specified in the **register_list** as follows:

'r1 r2 ... rn'

where **r1 r2** ... designate the 80386 CPU registers as shown in Table 9-2.

Table 9-2. 80386 Register Names

EAX	EBX	ECX	EDX	EBP	ESP	ESI	EDI
AX	BX	CX	DX	BP	SP	SI	DI
AH	BH	CH	DH				
AL	BL	CL	DL				
CS	DS	ES	FS	GS	SS		
EIP	EFLAGS						

CSID responds by displaying the name of the register followed by its current value. If you enter a carriage return, the register value does not change. If you enter a valid expression, CSID changes the register contents to the value of the expression. In either case, the next register is then displayed. This process continues until you enter a period or an invalid expression, or the last register is displayed.

Form 10 sets the Use Mode (see Section 3.2.1 for the assembler. The valid values are 10H (16-bit mode) and 20H (32-bit mode).

Form 11 sets the location of the debug process window's upper left corner to the coordinants specified by **row** and **column**. The row value determines the vertical location of the window; increasing the value moves the window further down on your screen. The column value determines the horizontal location of the window; increasing the value moves the window further to the right on your screen. Normally, the row value is 0 - 19H and the column value is 0 - 50H, but the maximum size for both can vary between systems.

Note: When you load a debug process via the command line (or LOAD command) with the W option, CSID creates a new virtual console (including a screen) for the debug process. It is this (the debug process's) virtual console/screen that is alterable by the SET WINDOW LOCATION command.

Form 12 sets the vertical size of the debug process window when the debug process is finished to a size specified by **row**. This resets the value specified by the CSID command-line option W.

Examples:

TBS

9.3 COMPARE COMMAND

The COMPARE command compares and displays the difference between two blocks of memory. The form is:

COMPARE s1,e1,s2

where **s1** is the starting address of the first block; **e1** is the ending address (last byte) of the first block, and **s2** is the starting address of the second block.

CSID displays any differences in the two blocks in the form:

a1 b1 a2 b2

where the **a1** and the **a2** are the addresses in the blocks; **b1** and **b2** are the values at the indicated addresses. If no differences are displayed, the blocks are identical.

Examples:

#compare 10000,101ff,40000
Compare 512 (200H) bytes of memory starting at 10000 and ending at 101FF with the block of memory starting at 40000.

#compare array1,0ff,array2
Compare a 255-byte array starting at address ARRAY1 with ARRAY2.

9.4 SEARCH COMMAND

The SEARCH command searches for a string of characters of values within memory. The form is:

SEARCH s,e,value

where **s** is the starting address to search and **e** is the ending address to search. **value** is an expression as defined in Section 7, including a string of 1 to 128 printable ASCII characters.

Examples:

#search 3006,31ff,0d0ah
search memory starting at 3006 and ending at 31FF for a two-byte value consisting of 0Dh (Carriage Return) and 0Ah (line feed).

#search 31ff,0d0ff,'ABCD'
search memory starting at 31FF and ending at 0d0ff for the character string: ABCD.

#search 31ff,0d0ff,41424344
search memory starting at 31FF and ending at 0d0ff for a four-byte value consisting of 41 (A), 42 (B), 43 (C), and 44 (D).

9.5 MOVE COMMAND

The MOVE command copies a block of values from one area of memory to another. The form is:

MOVE s,e,d

where **s** is the starting address of the block to move, **e** is the ending address of the block, and **d** is the address of the first byte of the area to receive the data.

Examples:

#move 3400,+9,4000
Move 9 bytes from 2400 to 4000.

#move array,+64,array2
Move 64H (100) bytes from ARRAY to ARRAY2.

9.6 FILL COMMAND

The FILL command fills an area of memory with a specified value. The form is:

FILL s,e,value

where **s** is the starting address of the block, **e** is the ending address, and **value** is an expression as described in Section 7.

Examples:

#fill 4100,413f,0 Fill memory at the current default display address from address 4100H through 413FH with the value 0.

#fill array,+0ff,0ff Fill the 255-byte block starting at ARRAY with the constant 0FFH.

#fill 3122,+100,'fillup'
Fill the block starting at address 3122H to address 3222H with the string constant 'fillup'.

9.7 ASSIGN COMMAND

The ASSIGN command assigns a command return value to a symbol name. The form is:

ASSIGN symbol_name command_name

where **symbol_name** is the name of a symbol, and **command_name** is the name of a CSID command.

Example:

#assign foo calculate 1000
Assign the value 1000 to the symbol named 'foo'.

#assign boo calculate foo+85
Assign the value 1085 to the symbol 'boo'.

9.8 CALCULATE COMMAND

The CALCULATE command performs basic arithmetic, logical, and relational calculations as described in Section 2.8.

The form is:

CALCULATE expression

where **expression** is an expression as defined in Section 7.

9.9 CLOSE COMMAND

The CLOSE command has two forms:

(1) CLOSE OUTPUT
(2) CLOSE SESSION

The CLOSE OUTPUT command closes the output redirection file set with the WRITE OUTPUT command (see Section 9.10).

The CLOSE SESSION command closes the user input session file (see Section 9.10).

Note: CSID automatically closes redirection and session files when you use the EXIT command.

9.10 WRITE COMMAND

The WRITE command writes the contents of a contiguous block of memory to disk. It also write macro files, user session files, and CSID output to disk.

The forms are:

(1) WRITE MACROS DELETE|APPEND,filespec,macro_name
(2) WRITE MEMORY DELETE|APPEND,filespec,s,e
(3) WRITE OUTPUT DELETE|APPEND,filespec
(4) WRITE SESSION DELETE|APPEND,filespec

where **filespec** is the name of the file you want to write to. If you specify DELETE, CSID deletes any existing file of the same name. If you specify APPEND, CSID appends the file to the existing file of the same name.

Form 1 writes the current definition of **macro_name** in command form to the specified file. FlexOS wildcards are allowed.

Form 2 writes the contents of a specific memory block. **s** is the starting address and **e** is the ending address of the block. If you do not specify the addresses, CSID assumes the first and last addresses from the files loaded with a READ command, causing all of the files loaded with READ to be written. If no file has been loaded with READ, CSID responds with an error message.

Use WRITE MEMORY for writing out files after patching code, assuming the overall length of the file is unchanged.

Form 3 allows you to direct your debugging session output to a printer or a disk file, in addition to the screen. **filespec** is the filename you want the output directed to, which can include physical devices. If **filespec** specifies a disk file, by default, CSID creates the output file under the current default directory.

Form 4 writes user commands input during a CSID session to the specified file. A session file can be read with READ COMMANDS for use in subsequent debugging sessions.

Examples:

#write memory test.386

Write to the file TEST.386 the contents of all previous READ FILE commands.

#write memory b:test.386,1000,3fff

Write the contents of the memory block 1000H through 3FFF to the file TEST.386 on drive B.

9.11 ASSEMBLE COMMAND

The ASSEMBLE command assembles 80386 mnemonics directly into memory. The form is:

ASSEMBLE {s}

where **s** is the address where assembly begins (the default is the current EIP). CSID displays the address, at which point, you can enter CASM assembly language statements.

When you enter a statement, CSID assembles it, places it in memory, and displays the address of the next available memory location. This process continues until you press the Carriage Return without entering any statement or after entering only a period.

CSID responds to invalid statements by displaying an error message and redisplaying the current address.

Wherever a numeric value is valid in an assembly language statement, you can also enter an expression. However, there is one important difference: while using the ASSEMBLE command, references to registers refer to register **names**, while elsewhere in CSID they refer to register **contents**. When using the ASSEMBLE command, you cannot reference the contents of a register in an expression.

Examples:

#a 1213 Assemble at address 1213.

00001213 **mov eax,128.**
Set EAX register to decimal 128.

00001216 **push eax**
Push EAX register on stack.

00001217 **call proc1**
Call procedure whose address is the value of the symbol PROC1.

0000121A **test byte [i\i], 80**
Test the most significant bit of the byte whose address is the value of the second occurrence of the symbol I.

0000121E **jz done** Jump if zero flag set to the location whose address is the value of the symbol DONE.

00001220 . stop assemble process.

9.12 MACROS

CSID supports a number of commands that allow you to define and use **macros**, which are single instructions that replace multiple instrutions.

By default, CSID uses a number of macros that are predefined in the file CSIDINIT.MAC, which is read whenever you invoke CSID. You can edit this definition file is desired, but you should keep it in the same directory with CSID, because CSID first searches for it in the current directory. If not found, CSID searches in the Home: directory, and if not found there, it searches in the System: directory.

These commands are:

(1) DEFINE macro_name
(2) DISPLAY MACROS {macro_name}
(3) ECHO ON|OFF
(4) IF command
(5) ELSE
(6) ENDIF
(7) BREAK_IF command

Form 1 defines a macro. The form is:

TBS

Form 2 displays the definition of the specified **macro_name**.

Form 3 toggles the display of macro information on and off.

Form 4 provides for conditional execution of CSID commands.

TBS

Form 5 specifies the start of an ELSE clause in a conditional statement.

TBS

Form 6 specifies the end of an IF clause in a conditional statement.

TBS

Form 7

TBS

End of Section 9

COMMON OBJECT FILE (COFF) FORMAT

FlexOS 386 executable files (filetype .386), object files output by the assembler or compiler (filetype .O), and library files (filetype .LIB) are all built according to the rules of UNIX™ System V COFF.

Figure A-1 illustrates the COFF format.

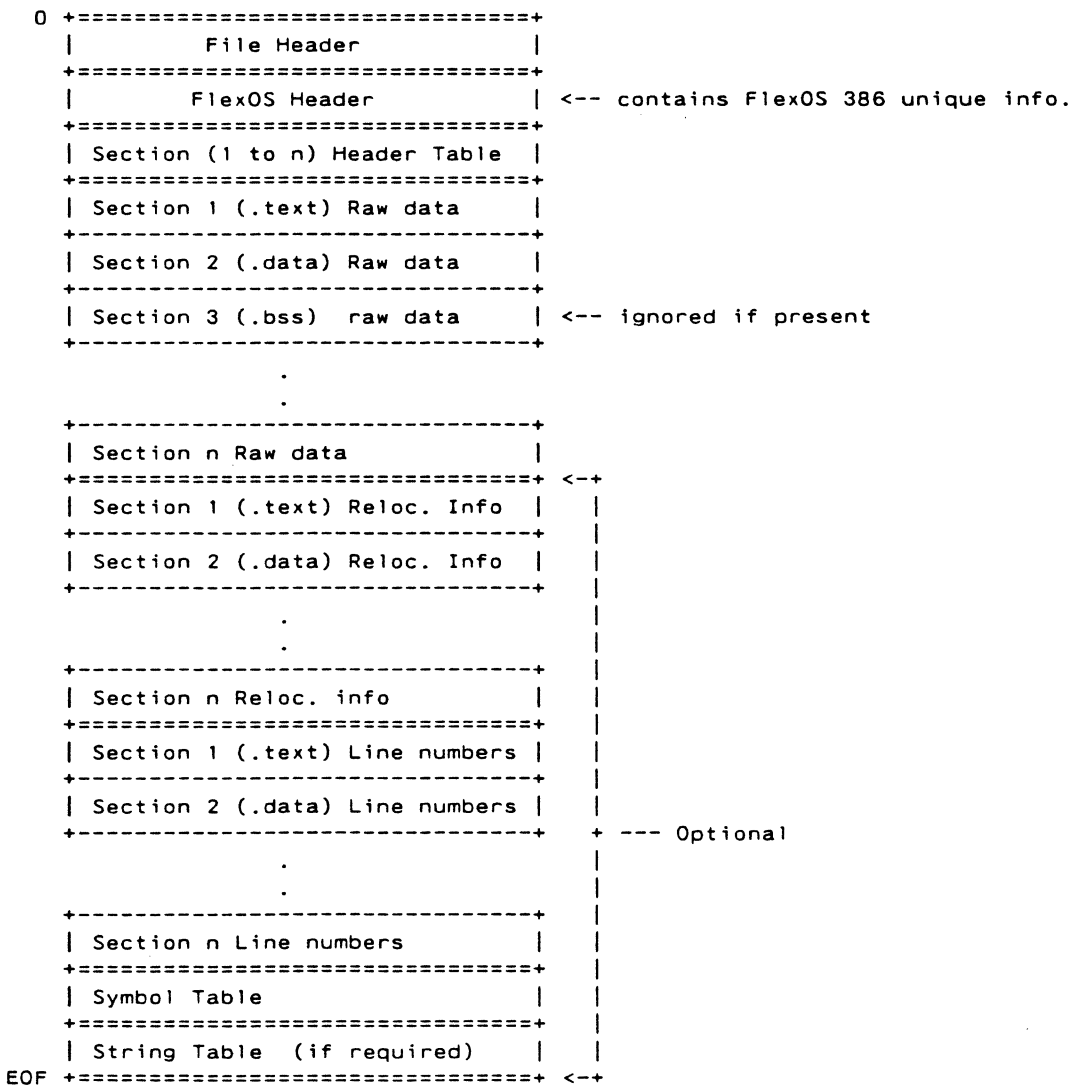

Figure A-1. Common Object File Format (COFF)

A.1. FILE HEADER

A 20-byte file header resides at the beginning of executables and object files. Figure A-2 illustrates the COFF file header.

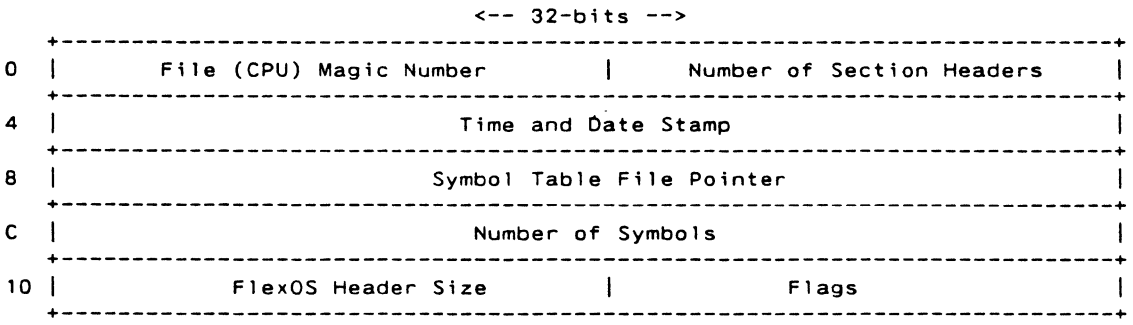

Figure A-2. COFF File Header

The following C code defines the file header:

```
struct filehdr
  {
  unsigned short        f_magic;        /* magic number                 */
  unsigned short        f_nscns;        /* number of sections           */
  long                  f_timdat;       /* time & date stamp            */
  long                  f_symptr;       /* file pointer to symtab       */
  long                  f_nsyms;        /* number of symtab entries     */
  unsigned short        f_opthdr;       /* sizeof(optional hdr)         */
  unsigned short        f_flags;        /* flags                        */
  };
/*
 *   Bits for f_flags:
 *   -----------------
 *  F_RELFL    relocation info stripped from file
 *  F_EXEC     file is executable  (i.e. no unresolved external references)
 *  F_LNNO     line numbers stripped from file
 *  F_LSYMS    local symbols stripped from file
 *  F_AR32WR   file has the byte ordering of an AR32WR machine (e.g. vax or I386)
 */
#define  F_RELFLG       0000001
#define  F_EXEC         0000002
#define  F_LNNO         0000004
#define  F_LSYMS        0000010
#define  F_AR32WR       0000400
#define  F_I386         F_AR32WR
#define I386MAGIC       0514        /* Intel 80386 */
#define FILHDR  struct filehdr
#define FILHSZ  sizeof(FILHDR)
```

The FlexOS 386 program loader requires the following bits to be set in the flags field of a .386 file:

- F_EXEC (the file contains no unresolved externals)
- F_I386 (the code consists of Intel 80386 instructions)

All other bits in the file header flags word are ignored.

A.2. FLEXOS HEADER

The optional FlexOS header immediately follows the COFF file header and contains information used by the FlexOS 386 program loader. The FlexOS header is ignored in CLINK input files, but CLINK initializes and inserts one into the output file.

Figure A-3 illustrates the FlexOS header.

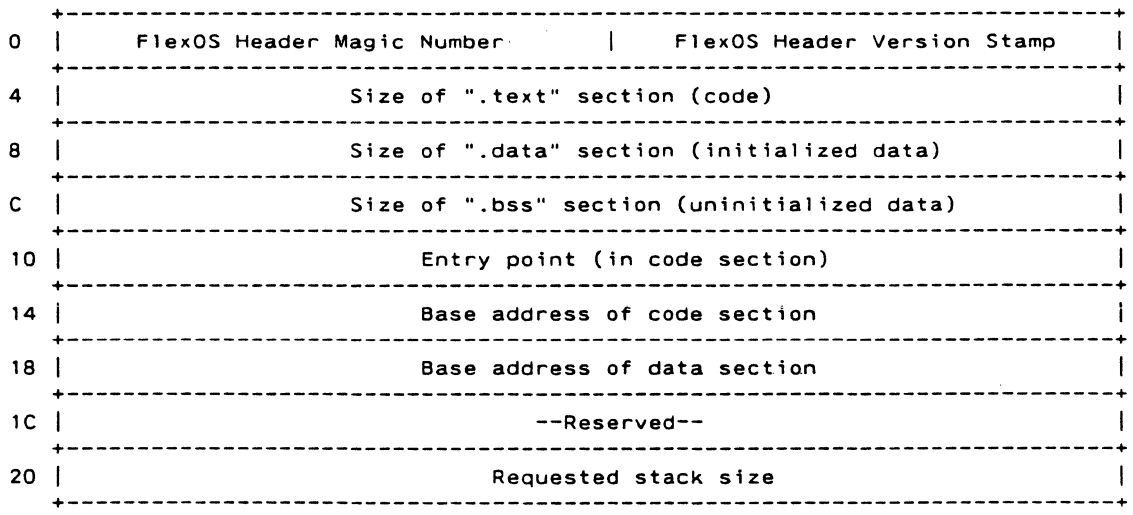

Figure A-3. FlexOS 386 File Header

The following C code defines the FlexOS file header:

```
typedef struct aouthdr {
        short   magic;          /* see magic.h                          */
        short   vstamp;         /* version stamp                        */
        long    tsize;          /* text size in bytes, padded to full
                                   word boundary                        */
        long    dsize;          /* initialized data "  "                */
        long    bsize;          /* uninitialized data "   "             */
        long    entry;          /* entry point in .text section         */
        long    text_start;     /* base address of text                 */
        long    data_start;     /* base address of data                 */

/* The following are FlexOS 386 specific: */
        long    res;            /* reserved */
        long    stk_size;       /* requested stack size in bytes*/
} EXTHDR;
```

The fields in the FlexOS header are defined as follows:

Magic number	700_O (Fast program load) The CODE and DATA sections are the only sections loaded. Space for the BSS section is allocated.
Version Stamp	This field normally identifies this load image as running under FlexOS 386. By default, CLINK inserts the FlexOS 386 signature: 3703_O or 1987_D or $7C3_H$ You can override the default with the CLINK **-VS** option.
Stack Size	CLINK pads this field to a 4Kb boundary. **Note:** The padded code section size plus the padded stack size must equal the data section base address for the Fast Load model.

A.3. SECTION HEADER

The **f_nscns** field in the COFF file header specifies the number of section headers that immediately follow the FlexOS 386 Header in the .386 file. Figure A-4 illustrates the section header.

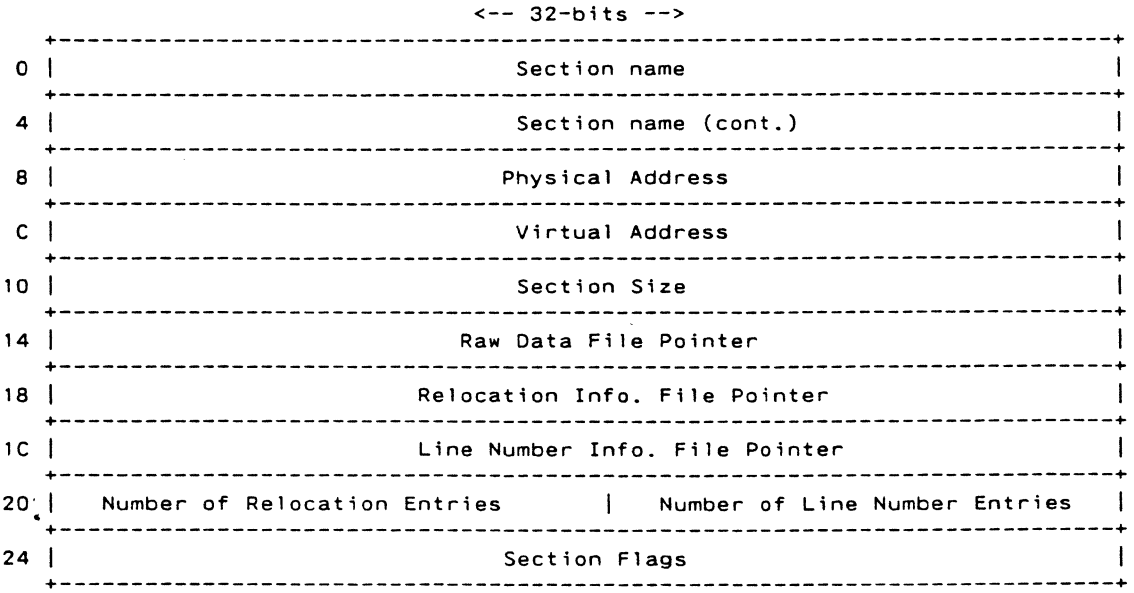

Figure A-4. Section Header Format

The following C code defines the COFF file section header:

```
/************* COFF file section header structure *************************/
struct scnhdr
  {
  char            s_name[8];      /* section name                          */
  long            s_paddr;        /* physical address                      */
  long            s_vaddr;        /* virtual address                       */
  long            s_size;         /* section size                          */
  long            s_scnptr;       /* file ptr to raw data for section      */
  long            s_relptr;       /* file ptr to relocation                */
  long            s_lnnoptr;      /* file ptr to line numbers              */
  unsigned short s_nreloc;        /* number of relocation entries          */
  unsigned short s_nlnno;         /* number of line number entries         */
  long            s_flags;        /* flags                                 */
  };

/************************************************************************/
#define SCNHDR  struct scnhdr
#define SCNHSZ  sizeof(SCNHDR)

/*
 * s_flags is used as a section "type"
 */

#define STYP_TEXT       0x20            /* section contains text only */
#define STYP_DATA       0x40            /* section contains data only */
#define STYP_BSS        0x80            /* section contains bss only */
```

The section physical address field is ignored under FlexOS 386.

A.4. RELOCATION ENTRY

Each section header includes two fields which refer to optional relocation information which may be present in the COFF file. **s_nlnno** provides the number of relocation items in the section. **s_lnnoptr** is a file pointer to the relocation entries. A COFF relocation entry points to a byte, word or long in the section's raw data, gives a symbol to which this data item is to refer, and specifies the type of reference.

Figure A-5 illustrates the format of a relocation entry.

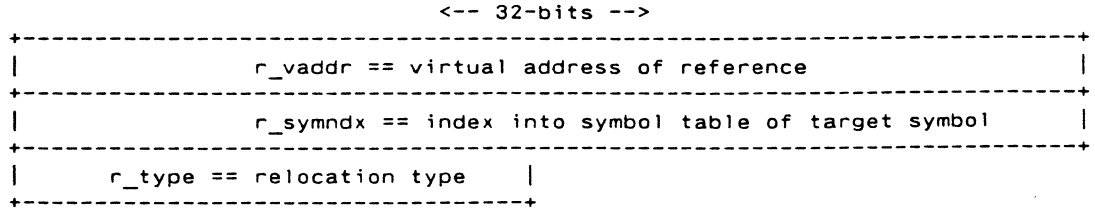

Figure A-5. Relocation Entry Format

The following C code defines the relocation entry:

```
struct reloc {
        long    r_vaddr;        /* (virtual) address of reference */
        long    r_symndx;       /* index into symbol table */
        unsigned short  r_type;         /* relocation type */
        };

/* relocation types for DEC Processors VAX 11/780 and VAX 11/750
 * and Intel 80386
 */

#define R_DIR16         001
#define R_DIR32         006
#define R_RELWORD       020
#define R_RELLONG       021
#define R_PCRBYTE       022
#define R_PCRWORD       023
#define R_PCRLONG       024

#define RELOC   struct reloc
#define RELSZ   10      /* sizeof(RELOC) */
```

The 80386 instruction set and CLINK support 5 relocation types:

R_DIR16 (0x0001)

The word at the reference address is to contain the direct 16-bit address of the symbol. If the relocated address of the symbol is greater than 0x0000FFFF, CLINK emits an error message. (Relocation types of R_RELWORD (0x0010) are also supported and handled in this way.)

R_DIR32 (0x0006)

The long at the reference address is to contain the direct 32-bit address of the symbol. (Relocation types of R_RELLONG (0x0011) are also supported and handled in this way.)

R_PCRBYTE (0x0012)

The byte at the reference address is to contain a signed distance from the reference to the symbol. If the relocated address of the symbol is out of range (+-128), CLINK emits an error message.

R_PCRWORD (0x0013)

The word at the reference address is to contain a signed distance from the reference to the symbol. If the relocated address of the symbol is out of range (+−32768), CLINK emits an error message.

R_PCRLONG (0x0014)

The word at the reference address is to contain a signed distance from the reference to the symbol. If the relocated address of the symbol is out of range (+−2147483648), CLINK emits an error message.

End of Appendix A

SAMPLE CASM SOURCE FILE

The following CASM source file demonstrates how to create a .386 file using CASM and CLINK.

```
; HELLO.A
title 'Sample .386 Program.'

; This program uses SVC's as follows:

;  1.  F_WRITE to stdout of "Hello world.".
;  2.  F_GET of process table.
;  3.  F_WRITE to stdout of process table.
;  4.  F_EXIT.
;
;  To generate this program, use the commands:
;
;     casm hello
;     clink hello -s

F_WRITE equ 8
F_GET   equ 0
F_EXIT  equ 25

PROCESS_TABLE      equ 0                ; table number
PROCESS_TABLE_SIZE equ 60               ; table size

        code                            ; specify .text section

; Program entry point:
; Hello, world.
        mov     eax,offset parm_blk     ; already setup for this call
        mov     ecx,F_WRITE
        int     221

; Get process table.
        mov     eax,offset parm_blk
        mov     long [eax],PROCESS_TABLE shl 8  ; put into "table" position
        mov     long 8[eax],0                   ; id = calling process
        mov     long 12[eax],offset table_buff  ; where to put it
        mov     long 16[eax],PROCESS_TABLE_SIZE
        mov     ecx,F_GET                       ; SVC number
        int     221

; Display the process table.
        mov     ecx,PROCESS_TABLE_SIZE/4
        mov     ebx,offset table_buff
proc_loop:
        push    ecx
        mov     ecx,4                           ; 4 bytes per line
pr_lp:
        push    ecx
        mov     al,[ebx]
        push    ebx
        call    display_hexbyte
        pop     ebx
        inc     ebx
```

```
        pop     ecx
        loop    pr_lp

        push    ebx                             ; save byte ptr
        call    print_crlf
        pop     ebx
        pop     ecx                             ; line ptr
        loop    proc_loop
eject

; Exit to OS.
        xor     eax,eax                         ; return code = 0
        mov     ecx,F_EXIT
        int     221

; Subroutines:

display_hexbyte:                        ; Entry:        al = binary byte
        push    eax
        shr     al,4
        mov     ebx,offset hex_table
        xlat
        push    ebx
        call    char_out
        pop     ebx
        pop     eax
        and     al,0fh
        xlat
        call    char_out
        mov     al,' '
;       call    char_out
;       jmp     char_out

char_out:                               ; Entry:        al=ascii char
        mov     char,al
        mov     ecx,F_WRITE
        mov     eax,offset write_buff
        int     221
        ret

print_crlf:
        mov     al,0dh
        call    char_out
        mov     al,0ah
;       call    char_out
        jmp     char_out
eject

; HELLO data section:

        data                            ; specify .data section

parm_blk        rl      0
p_0             dl      0               ; sync, options=0, flags = 0
p_1             dl      0               ; no swi
p_2             dl      1               ; stdout (already opened)
p_3             dl      offset string   ; what I'm writing
p_4             dl      length string   ; how much I'm writing
p_5             dl      0               ; offset into "file"
```

```
string          db      'Hello world.',0dh,0ah

table_buff      rb      PROCESS_TABLE_SIZE      ; Where OS puts table

write_buff      dl      0,0                     ; single character F_WRITE
                dl      1                       ; parameter block
                dl      offset char
                dl      1
                dl      0
char            rb      1

hex_table       db      '0123456789ABCDEF'

        end

; End of HELLO.A
```

End of Appendix B

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z