SYSTEM DEVELOPMENT DOCUMENTATION

Document Revision 3
December 1995

DM_ASM and DM_COFFLINK

Drive Manager
Macro Assembler and Linker User's manual

adaptec®

NOTICE

This manual describes the proprietary COFF (Common Object File Format) macro assembler (DM_ASM) and COFF linker (DM_COFFLINK) for the DSP assembly language used with Adaptec's AIC-44XX DMC. DMC is an acronym for Drive Manager IC containing a DSP core (PINE™) and proprietary circuitry on a single IC. The words PINE and DM (Drive Manager) may, at times, be used interchangeably in the text.

The information contained in this document is subject to change without notice.

Adaptec and the Adaptec logo are registered trademarks of Adaptec, Inc. PINE, PINEASM, and COFFLINK are trademarks of DSP Semiconductors USA, Inc. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. All other trademarks used are held by their respective owners.

Table of Contents

This page intentionally left blank.

SECTION 1
Introduction

1.1 General Description

This manual describes the proprietary COFF (Common Object File Format) macro assembler (DM_ASM) and COFF linker (DM_COFFLINK) for the DSP assembler language. The Drive Manager Assembler will be referred to as DM_ASM and the Drive Manager Linker as DM_COFFLINK. The information in this manual is updated to correspond to the DM_ASM version 6.2, and to DM_COFFLINK version 6.2. Finally, this manual will briefly describe how to generate programmable load files, linker maps, and symbol tables.

This manual assumes that the reader is familiar with assembly programming and with the DMC instruction set. It explains the installation procedures, the various invocation parameters, all assembler directives, the linker script files format and options, some programming hints, error messages and restrictions or limitations.

1.2 Related Documents

DSP Core Programmer's Manual, PN: 700175-011
DM_ASM and DM_CoffLink User's Manual, PN: 700174-011
DM_DBG Programmer's User's Manual, PN: 700176-011
AIC-4420 Drive Manager Chip Data Sheet, PN: 700211-011
AIC-4420 Drive Manager Chip ROM Code User's Guide, PN: 700185-011

1.3 What's New

Below is a short list of the changes in version 6.2.1 versus 6.2.0. For the most complete and up-to-date information, see the release notes.

1.3.1 Assembler

1. Octal Constants

   The new assembler allows the use of various numeric representations, including the new OCTAL representation, consisting of any string of the numeric characters 0, 1, ..., 7 starting with a zero (0). For example, 017 is the octal representation of fifteen, not to be confused with 17, which is the decimal representation of seventeen.

   By default, in version 6.2.1 of the assembler, the octal format is disabled, to allow compatibility with old code. To enable the new octal format, add the invocation the "-octal" flag, i.e.

   ```
   pineasm6 -octal <assembly_file.asm>
   ```

```
pineasm6 -octal <assembly_file.asm>
```

2. Improved Restriction Checking

   The new assembler, version 6.2.1 correctly flags certain architecture restriction violations related to the "bkrep" instruction, previously assembled and left undetected in version 6.x.

3. Undefined and Undeclared Label References

   The assembler now issues a warning whenever an undefined and undeclared label is referenced in a particular module. To get rid of the warning, an explicit .EXTERN or .GLOBAL declaration of the label must be preceded the usage of the label.

1.3.2 Linker

1. Improved Coff Section Map

   As of version 6.2.1, the coff section map is ordered according to the section addresses. Furthermore, the section names are now nicely aligned to create a much more readable and useful map. By default, the map section does not include symbol relocation information.

2. Improved Coff Symbol Table

   As of version 6.2.1, the coff symbol table is ordered according to the symbol name and provides the absolute (relocated) address of all the symbols after linking all sections.

3. Long File Names (Unix)

   Version 6.2.1 fixes a bug encountered in the Unix version of version 6.2.0 of long library file names (including the full path) in the linker script file.

4. Positive Numeric Offset Bug

   Version 6.2.1 fixes a bug encountered in version 6.2.0, when positive numeric offsets were used, e.g.

   ```
   move @ Label+2, r0
   ```

   or

   ```
   move @ Label+2, r0
   ```

   The correct value is now used for patching the opcode in the object

5. Library Files

   The Linker has been fixed when working with library files.

6. The linker now correctly handles the "next(...)" directives in overlays defined in the linker script file.

SECTION 2
Installation

2.1 Package Contents

The list of files supplied on the DM_ASM / DM_COFFLINK software package diskette includes:

Documentation Files:

READASM.TXT- initial instructions, latest information

Batch Files:

COFF2DMC.BAT	- creates DMC format file for ROM mask creation
COFF2HEX.BAT	- creates a PROM burn file in Intel hex format
AA.BAT	- invokes macro preprocessor, DM_ASM and listing enhancer
LINK6.BAT	- invokes DM_COFFLINK linker
L.BAT	- invokes DM_COFFLINK and prepares RS-232 serial port download file (using ROM software)
FL.BAT	- invokes DM-COFFLINK and prepares RS-232 serial port download file for rogramming flash EEPROM (using flash support software)
LINK2ROM.BAT	- invokes linker and generates ROM format file for mask creation

Executable Files:

COFFLNK6.EXE	- PINE COFF object linker
COFFUTIL.EXE	- utility to extract COFF in HEX format (used by COFF2HEX.BAT)
ERROR.COM	- utility used by batch files to return error level
HEX2DMC.EXE	- utility to convert HEX file to DMC file
INTELHEX.COM	- utility used by COFF2HEX.BAT to generate PROM burn file(s)
MPP.EXE	- PINE macro preprocessor
PINEASM6.EXE	- PINE macro assembler
SORTHEX.EXE	- utility used to sort COFF contents
DOWNLOAD.EXE	- creates TXT file for RS-232 serial port download into the DMC
PINEABS.EXE	- creates ABS listing with absolute addresses
COFFLIB.EXE	- COFF object library archiver (ver 1.0)
HEX2ROM.EXE	- utility to convert HEX file to ROM file

2.2 Installation Instructions for New Users

The DM_ASM and DM_COFFLINK programs require MS-DOS version 5.0 or higher. For a first time installation, either 1) perform the following four steps, or 2) run the INSTALL.BAT file on the installation diskette and then perform steps 3 and 4 only:

1. Create Directory

   It is suggested that the user install all DM_ASM/COFFLINK files in a directory named C:\TOOLS\DMC. The name of the directory is not important, but it must be set as an environment variable named DMCTOOLS (see item 3 below). Change to this directory:

   ```
   MD  C:\TOOLS\DMC
   CD  C:\TOOLS\DMC
   ```

2. Copy Files

   Copy or uncompress all files in the diskettes' root directory to C:\DMC\TOOLS directory:

   ```
   COPY  A:\*.*
   PKUNZIP  A:\*.ZIP
   ```

3. Modify AUTOEXEC.BAT File

   Add the following line to your AUTOEXEC.BAT file:

   ```
   SET PINETOOLS=C:\TOOLS\DMC
   SET DOS4G=QUIET
   ```

   Next, make sure that an environment variable TEMP has been defined in your AUTOEXEC.BAT file. It must be set to a directory that will be used for scratch files. A RAM disk can be used for this purpose.

   NOTE: the TEMP directory name can not contain a trailing backslash (\), i.e. when it is the root directory of a device, e.g.

   ```
   SET TEMP=E:\TEMP      OK
   SET TEMP=E:           OK
   SET TEMP=E:\          Wrong!
   ```

   Change the PATH environment variable to include the DMCTOOLS directory:

   ```
   PATH=%PATH%;%PINETOOLS%
   ```

4. Modify CONFIG.SYS File

   In order to ensure ample environment space, your CONFIG.SYS file must specify COMMAND.COM and the amount of environment space, (the default of 256 is normally not sufficient), e.g.

   ```
   SHELL=C:\COMMAND.COM /P /E:1024
   ```

The installation procedure is now complete, and you can start using DM_ASM/COFFLINK COFF macro assembler and linker.

NOTE: The version 6 tools use a 32-bit DOS extender technology to allow use of extended memory.

SECTION 3
DM_ASM Description

3.1 General Notes

DM_ASM is a case-sensitive COFF macro assembler that fully supports the DSP assembly language. It permits dynamic memory allocation at link time and has complete DSP programming restrictions checking. DM_ASM prepares the object for full symbolic debugging with the DM_DBG software. It has C-like operators and conventions that allow easy development of code and data structures.

COFF object files created by the assembler are linked by DM_COFFLINK via a linker script file into a executable COFF load file. This load file can be loaded by the DM debugger software for symbolic emulation or simulation.

COFF makes modular programming simple because it enables the programmer to think in terms of blocks of code and blocks of data, referenced hereafter as segments or sections. Assembler and linker directives enable the programmer to easily create and relocate sections. Labels are referenced using a segment name followed by a dot and an offset in that segment, similar to the way that labels are identified in a typical C debugging environment. All segments are of either CODE or DATA type. The load file can consist of any arrangement of either CODE or DATA segments. All segment names and labels defined as "external" are automatically stored in the symbol tables of the COFF files, so that symbolic debugging can be performed in the debugger once the COFF executable has been loaded into memory.

The DM macro assembler is comprised of three parts: the macro preprocessor, the main program that analyzes the assembly instructions and a post-processor for listing enhancement. The macro preprocessor is the DOS executable program MPP.EXE. The main program is the DOS executable file PINEASM6.EXE. The post-processor is the DOS executable LST.EXE.

The macro preprocessor has the following purposes:

1. Merges included files.
2. Replaces macros and equated strings.
3. Filters the relevant portions of the input file in case of conditional assembly.
4. Prepares line number information for the assembler.

The main program has the following two passes:

1. Assembly pass - where syntax is checked and the COFF object file is built.
2. Restriction pass - where DSP architectural restrictions are checked.

3.2 DM_ASM Invocation

3.2.1 Batch File Invocation

The normal way to invoke the DM macro assembler is via the batch file AA.BAT, where the argument is the base name of the source file, i.e., without the mandatory .ASM extension. From the DOS command line type:

```
AA BaseFileName
```

If you receive the DOS error message "out of environment space", increase the environment space in your CONFIG.SYS file (see installation instructions in Section 2).

3.2.2 Output Files

The outputs from running DM_ASM with AA.BAT are an object file (.O extension) and a listing file (.LST extension). In the%TEMP% directory, a temporary MPP.TMP file is created, which can be viewed for debugging problems occurring during the macro preprocessing stage (see also the general notes above).

3.2.3 Using Make Files

There are two ways to use MAKE utilities to run DM_ASM and DM_ COFFLINK. If you have the latest versions of the MAKE utilities from Microsoft, you can run a "dynamic make", i.e., a MAKE that has the ability to swap to extended memory or to swap to disk. In this case do not write CALL before batch file commands. If you have a more restricted MAKE version, you can run a "static make", i.e., a MAKE that produces a batch file of what it would have executed dynamically, and then execute the batch file after the MAKE is finished. In a static MAKE, you must write CALL before batch file commands. In Microsoft MAKE utilities, static makes are obtained when invoking with the -N option. Note that if your conventional memory is limited, it is recommended to use a static make.

3.2.4 Command Line Invocation

The MPP.EXE preprocessor can be invoked directly from the DOS command line:

```
mpp [options] sourcefile > outputfile
```

Options:

-iPathName	To look for included files in the provided path if not found in current directory. Multiple directories can be given, separated by ";" or ",".
-s	To filter out all statements falling under false assembly condition. By default, these lines are printed as comments.
-c	To filter out all user's comments.
-w	To print warning messages in output file.
-h	To print a help message.
-o FileName	To force a particular output filename (replaces the "> outputfile" part)

NOTES:

1. Uppercase letters can also be used for selecting the options (-I,-M,-S,-C,-W).
2. An environment variable named MPP can be set with a DOS command to specify a path for searching included files, e.g. "SET MPP=c:\user\include;x:\pine\inc". If both environment variable MPP is set and the -i option is used, then the environment variable is ignored. By default, the active directory is always searched first.
3. MPP returns a DOS error code of 1 upon detection of any kind of error.

The PINEASM6.EXE main program can also be invoked directly from the DOS command line:

```
pineasm6 [options] < sourcefile > listingfile
```

Options:

-iPathName	To look for included files in the provided path if not found in current directory. Multiple directories can be given, separated by ";" or ",".
-s	To filter out all statements falling under false assembly condition. By default, these lines are printed as comments.
-c	To filter out all user's comments.
-w	To print warning messages in output file.
-h	To print a help message
-o FileName	To force a particular output filename (replaces the "> outputfile" part)

NOTES:

1. PineASM returns a DOS error code of 1 if any errors occur.
2. If using the interactive mode (option -S), activate the assembler by entering a single or double ^Z (control+Z) character sequence.

3.3 Instruction Set Syntax

The instruction set syntax is summarized in appendix A of this manual. The following list specifies programming conventions assumed by the DM_ASM COFF macro assembler, not mentioned in the architecture specification.

CONVENTIONS:

1. The COFF assembler is case-sensitive. Opcode mnemonics, register names, and flag names are all lower case. Assembler directives are all upper case. Directive, mnemonic, register, and flag names are reserved words. User labels should not conflict with them.
2. The hexadecimal and binary numeric formats are C style, e.g. 0x1234 and 0b1010.
3. The syntax used for offset addresses, i.e. the location counter relative jump address used in **brr** and **callr** instructions, is the following:

```
$
$ + NumericExpression
$ - NumericExpression
NumericExpression + $
```

   Example:

```
brr  $+1
```

4. There are two immediate value operators, # and ##, for short values (8 bits) and long values (16 bits) respectively. The following instructions can accept both forms: **mov, add, sub, and, or, xor, cmp**. For these instructions, the assembler will automatically use the long format for immediate constants that can not be represented by 8 bits. All other instructions that can accept an immediate value operand, accept only the short form. They must be specified with #. Instructions that allow only long immediate values should have the ## operator prefixed to their operands.

   Examples:

```
mov #0, r0        ; one word instruction
mov ##0, r0       ; two word instruction
mov #0xffff,r0    ; one word instruction, sign extension ignored
mov #0x7fff,r0    ; two word instruction
mov #0,a0         ; illegal one word instruction (see architecture
                  ; specification)
mov ##0,a0        ; two word instruction
mov #0,a0l        ; one word instruction
mov #0,a0h        ; one word instruction
```

5. The movp instruction is different when the first operand is the accumulator. The low accumulator must be explicitly specified, e.g.

```
movp (a0l), r5    ; correct syntax
movp (a0), r5     ; architecture specification syntax - illegal format
```

   Note that the **movp** syntax conforms with **calla**.

6. To ensure full architectural restriction checking associated with the **bkrep** instruction, a **bkrep** block must be completely contained within a single COFF section, i.e.
   a) No .CODE, .DATA or .ORG directives are allowed inside a **bkrep** block.

b) The second operand of the **bkrep** instruction, which contains the terminating address of the block, can contain only a forward reference to a temporary or permanent label declared in the same COFF section.

7. To ensure full checking of relative branches and calls, the **brr** and **callr** instructions can branch to a (relative) address only within the current COFF section.
8. Comments can be added in 2 ways: on a single line (e.g. after an instruction), by preceding it with a semicolon (;), or on multiple lines as in C programs (/* comment */).

3.3.1 Full Name vs. Simple Labels

For compatibility reasons with previous versions of the assembler and linker, permanent labels are of either two types: "full name" or "simple". The type depends on the section (segment) in which they are defined.

Labels that are defined in a .CODE or .DATA section, are "full name" labels. They inherent the name of that section as a prefix to the label, separated with the dot (.) operator. A reference to such a label, needs the specification of the full name, i.e. "segment_name.label_name". Inside a .CODE or .DATA section, if a reference is made to a label, it is automatically prefixed with the section name, unless an explicit section name is prefixed by the programmer. Every time a new .CODE or .DATA section is declared, a new prefix is active which is automatically given to label definitions and added to label references in that section. By using the .USE directive, the default prefix to label references can be changed until a new .CODE or .DATA section is declared, or a new .USE declaration is made. The directives .PUSHSEG and .POPSEG can be used inside .CODE and .DATA sections, to temporarily change the active segment for prefixing, and restoring the previous active segment, without knowing the section name, which is useful for macros that can be invoked in different segments. Note that "full name" labels can have the same label name in the same module if they are defined in different segments.

The "simple" labels on the other hand, are labels that are defined in a .CSECT or .DSECT section. Inside a particular module (file) they must have a unique name, across both program (code) and data memory spaces. By default, they are local, unless the .PUBLIC directive is used to declare them global, in which case they may not be redefined in any other section or module. Simple labels do not inherent any segment prefix and are referenced just by their name.

To refer to a "full name" label inside a .CSECT or .DSECT section, just add the appropriate segment prefix. To refer to a "simple" label from within a .CODE or .DATA section, you must instruct the assembler not to automatically add the segment prefix by first issuing a ".USE 0" command. In this case, this mode will be in effect until a new .CODE, .DATA or .USE directive is used. See the description of the various directives for more details.

3.4 Arithmetic and Logical Operators

The arithmetic and logical operators are a subset of the C language operators. All of the operators are effective at assembly time on resolvable constants. The order of expression evaluation is the same as in C.

The following are the supported operators:

+	Addition operator
-	Subtraction operator
/	Integer division operator
*	Multiplication operator
%	Modulo operator
&&	Logical-And operator
||	Logical-Or operator
&	Bit-And operator
|	Bit-Or operator
^	Bit-Xor operator
>>	Arithm. shift-right operator (sign ext)
<<	Bit shift-left operator
Unary +	Positive operator
Unary -	Sign change operator
Unary ~	Bit complement operatoR
Unary !	Logical not operator
(expr)	Group operator
==	Equal test operator
!=	Not equal test operator
>=	Greater than or equal test operator
>	Greater than test operator
<=	Less than or equal test operator
<	Less than test operator
expr ? v1 : v2	Conditional operator (if expr then v1, else v2)

NOTE: The C paste operator (##) and string operator (#) are not supported, because they are used to specify long or short immediate values in immediate addressing (see Section 3.5 below). The paste operator is described in Section 3.6.

3.5 Assembler Operators

Operator

The long immediate operator is both an assembly-time and link-time operator which accepts a word (16-bit) value.

Example: `mov ##label+1, r0`

Operator

The short immediate operator is both an assembly-time and link-time operator which accepts a byte (8-bit) value.

Example: `mov #0, r0`

$ Operator

The location counter operator is a link-time operator which represents the address of the next instruction. An operand expression may not include both a label and a '$' together.

Example:
```
brr $-1             ; branch to self
mov #$+label, r0    ; illegal
```

: Operator

The (:) is the label definition operator. It is both an assembly-time and link-time operator. The offset name is the symbolic name of the label with respect to the current segment. Note that the directives .USE, .PUSHSEG, and .POPSEG, have no effect on the current segment. The unabbreviated reference to this label is "CurrentSegmentName.OffsetName." The abbreviated reference to this label is "OffsetName." It is affected by the use of the directives .USE, .PUSHSEG, and .POPSEG.

NOTE: The trailing colon (:) is **mandatory**. The maximum length of an offset name is 31 characters. It must start in the left-most column with a letter (lower or upper case) and may include any letters, digits or underscore (_) symbols. Preceding as well as trailing blanks and tabs are ignored. Labels are case sensitive.

Example:
```
My_Label2:      mov  #0,y              ; label definition
                brr    My_Label2, eq   ; label reference
```

%...: Operator

The (%...:) is the temporary label definition operator. It is an assembly-time operator. The scope of a temporary label is only within the current COFF section, i.e., the current instance of the current segment. The current section terminates with the .CODE, .DATA, or .ORG directives. Temporary labels must be used only in the **bkrep** instructions, to ensure proper restriction checking. Temporary labels may be used by all other branching instructions, for example when the programmer is exhausting his label naming creativity, or when multiple program pieces exist that are branched to upon the same cause or condition.

NOTES:

1. Temporary labels are not symbolically disassembled by the debugger.
2. The temporary label must be followed by an instruction before the end of the COFF section in which it is defined, otherwise it will be ignored.

>%Label and <%Label Operators

These operators are the assembly-time forward and backward temporary label reference operators respectively. The closest forward or backward reference is used respectively. Temporary label references can not be prefixed with a segment name. The scope of a temporary label is only within the current COFF section, i.e., the current instance of the current segment. The current section terminates with the .CODE, .DATA, or .ORG directives.

NOTES:

1. By default, references to temporary lables are assumed to be forward.
2. The reference to the last address of a **bkrep** loop must be a temporary label.

Example:
```
        add r0, a0
        br >%Ok, lt
        ...
    %Ok:
        add r1, a0
        br >%Ok, gt
        ...
    %Ok:
```

. Operator

The (.) dot operator, or segment prefix operator, allows for fully specified "full name" permanent label references. It is an assembly-time operator. Offset names are only unique within segments, i.e., the same offset name may be defined in many segments. An offset name is unique only when prefixed with its segment name. By default, "full name" label references (label references inside .CODE or .DATA sections) that do not contain an explicit segment prefix, use the prefix of the current .CODE or .DATA segment, i.e., the segment where the label is referenced (and not necessarily where it is defined). The .USE, .PUSHSEG, and .POPSEG directives affect the default behavior of

unprefixed "full name" type of labels (see section 3.4 above). Names, numbers or numeric expressions can be used for offsets (which might be useful for addressing arrays). The dot operator can not be used with "simple" labels. Names, numbers or numeric expressions can be used for offsets (which might be useful for addressing arrays).

Examples:

```
mov #SegName.OffsetName1, r0
mov #SegName.15, r0
mov #SegName.(3*5)
```

The proper way of referencing the base address of a segment is SegName.0, where only a segment name is required as an external reference (see section 3.7 below) and no offset names are required. SegName.0 must be declared as external, e.g.

```
.EXTERN SegName.0
```

Even though the assembler does not currently support a type definition or structure definition directive, with the use of simple macros, one level structures can be defined. This can easily be used to declare the same structure in multiple segments (see also section 5).

@ Operator

The (@) at operator, or modulo-256 operator, allows for automatic label references in direct addressing mode. This is a link-time operator. Symbol names preceded with this operator, will be treated as direct memory addresses by the linker, i.e. their final (relocated) address will be truncated to 8 bits by modulo 256 operation on the address value.

Examples:
```
mov @SegName.OffsetName1, r0
mov a0h, @SegName.Label
```

FRACT(number,bits) Operator

The fract operator is an assembly-time operator which calculates the 16 bit integer value representing the floating point operand in a user supplied fractional representation. The fractional representation is specified by indicating the number of bits to the right of the floating point.

Examples:
```
mov  ##FRACT( 0.25, 15), r0 ; translates to  mov ##0x2000, r0
mov  ##FRACT( 3.5, 12), r1  ; translates to  mov ##0x3800, r1
```

In the first example, 1 bit is used for the sign of the number and 15 bits are allocated for representing the fraction 0.25. In this format, the range of values that can be used is from -1.0 to +1.0 (not including the limits). In the second example, 1 bit is used for the sign of the number, 3 bits are allocated for the integer part of the number and 12 bits are used for the fraction part of the number. For more examples and programming hints, see also Section 5.7.

IMMEDOFFSET Operator

The immediate offset operator is an assembly-time operator which calculates the offset of a label from within the segment in which it is defined. The label can not be an external or forward reference. This operator can effectively be used to calculate the number of consecutive variables. defined in a long data segment. See also Section 5.3 for more examples on how to use this operator.

Example: `mov #IMMEDOFFSET Segment.Offset, r0`

INCODE Operator

The INCODE operator is a link-time operator. It preceeds a label and instructs the linker to use the address of the label residing in the program space. This operator is used when creating down-loadable programs. In this case, a section might be linked in both the program and data space. Labels in this section are thus defined twice, in both the data space and program space. By default, all label references used in call(r) and br(r) instructions refer to the program space, and all other label references are assumed to point into the data space. In case a down-loadable program is created, a label reference used in an instruction which is not call(r) or br(r) might be a reference into the program space, in which case it must be preceeded with the operator INCODE.

Example:

```
mov ## INCODE MyTable, r4   ; take address of MyTable in the program space
movp (r4)+,(r0)+
```

OFFSET Operator

The offset operator is a link-time operator which calculates the offset of a label from within the segment in which it is defined. If the label on which this operator is used resides in a data segment that is located on a page boundary (specified at link time), this operator can be used for direct addressing mode. Upon linking, the object code that corresponds to the label is patched to reflect the distance between the final label address and the final address of the start of the segment in which this label is defined.

Example: `mov #OFFSET Segment.Variable, r1`

See also Section 5.6 for more examples on this operator and how it can be used for direct memory addressing.

PG Operator

The PG(Symbol) link-time operator finds the 256-word memory page of the symbol. It is equivalent to using SHR(Symbol, 8). This link time operator enables the programmer to load the processor's page register with the **lpg** instruction, without worrying where the symbol is eventually located by the linker.

Example:

```
lpg  #PG( Segment.Offset )
```

NOTES:

1. For trouble free data memory accesses using the efficient short direct addressing mode, the programmer must guarantee via the linker that his data segments are aligned on page boundaries (using the **align** linker option) and that the data segments do not exceed the physical page size of 256 words (using the **inpage** linker option). If these conditions are not met, the programmer must change the page bits each time according to the data variable being accessed. See also section 5.6 for more explanations and examples.
2. The PG operator can be used with arithmetic expressions of the type +const or -const.

SHR Operator

The SHR(Symbol,nBits) link-time shift-right operator executes a bit shift-right. It differs from ">>" which is an arithmetic shift-right assembly-time operator. This link time operation allows efficient loading of the preocessor's page register via the LPG #immediate instruction.

Example: `lpg  #SHR( Segment.Offset, 8)`

NOTES:

1. The programmer must guarantee via the linker that the segment is aligned on the proper boundary.
2. nBits must comply with: 0<= nBits <= 15
3. See Section 5.6 for more examples and how this operator can be used for efficient direct memory addressing.

SIZEOF Operator

The SIZEOF(SectionName) link-time operator calculates the size of a section. This link time operator helps creating efficient code when the size of a section is a parameter. Examples are initialization programs that need to inialize all the variables allocated to a particular data section.

Example:

```
clr   a0
rep   #SIZEOF( MyData )-1
mov   a0l, (r1)+
```

NOTE:

The SIZEOF operator can be combined with simple arithmetic expressions of the type+const8 or -const8 (where const8 represents an 8 bit signed number) as shown in the above example.

3.6 Macro Preprocessor Operators

The macro preprocessor supports the following string operator:

' Operator

The paste operator implemented as a single quote ('), enables pasting in C-like "define"s, e.g.:

```
.EQU  Index  10
.EQU  String Abc
mov   #Label'Index, r0  ; which translates to mov #Label10, r0
mov   String'Index, y   ; which translates to mov Abc10, y
```

3.7 Assembler Directives

Assembler directives supply program data and control the assembly process. They allow partitioning of code and data into sections, allocation and initialization of memory, definition of global variables, conditional assembly and control the appearance of the listing. For all directives (except the DW directive) the first non-blank character of the line must be a dot (.). All directives must be specified in upper case (as opposed to the instruction mnemonics). The following is a complete list of the assembler directives supported by DM_ASM.

.CODE [SegmentName]

This directive defines the start of a code segment. If no segment name is supplied, the default code segment "CODE" is used. The .CODE directive creates a new COFF section that can be linked by the linker with other .CODE, .CSECT, .DATA or .DSECT sections into the processor's program (code) or data memory space. When a new COFF section of this type is created, the previous temporary symbol table is deleted, and the current .USE segment name is set to the argument of the .CODE directive. All labels defined in the newly created COFF section, are of type "full name". This means that the segment name is implicitely attached to all the labels, and that to refer to such a label, it is necessary to specify the full name (e.g.SegmentName.Label).

Example:

```
.CODE       MyCodSeg
```

NOTES:

1. The DW (Data Word) directive described below can not be used inside a .CODE section. To create a table of constants in the program space, link an appropriate .DATA or .DSECT section into the program memory space, by specifying the .DATA or .DSECT section name in the linker script file together with the program's code sections.
2. .CODE segment names are automatically PUBLIC and must be unique across all modules, all code, and all data segments, regardless of whether the segment is only used privately within a module.
3. .CODE segments may be split within a particular module or in different modules, creating multiple sections of this segment. These sections are then glued together at link time to form one code segment according to the linking algorithm (see sections 4.4 and 5.5 for details).

.CSECT SectionName

This directive defines the start of a code section. The .CSECT directive creates a new COFF section that can be linked by the linker with other .CODE, .CSECT, .DATA or .DSECT sections into the processor's program (code) or data memory space. When a new COFF section is created, the previous temporary symbol table is deleted. All labels defined in the newly created section, are of type "simple". This means that no segment name is attached to labels, and that to refer to such a label, it is enough to specify the label name. The .CSECT directive must be followed by a section name.

Example:

```
.CSECT      MyCode
```

NOTES:

1. The DW (Data Word) directive described below can not be used inside a .CSECT section. To create a table of constants in the program space, link an appropriate .DATA or .DSECT section into the program memory space, by specifying the .DATA or .DSECT section name in the linker script file together with the program's code sections.
2. .CSECT segments may be split within a particular module, or in different modules,creating multiple sections of this segment. These sections are then glued together at link time to form one code segment according to the linking algorithm (see sections 4.4 and 5.5 for more details).

.DATA [SegmentName]

This directive defines the start of a new data segment. If no segment name is supplied, the default data segment "DATA" is used. The DATA segments contain data definitions (not instructions). By default, the linker maps the .DATA segments into the processor's data space regardless of whether it contains initialized data. The linker can be instructed, however, to map .DATA segments into the processor's program (code) space, for example, to include constant tables or filter coefficients. When a new COFF section of type .DATA is created, the previous temporary symbol table is deleted, and the current .USE segment name is set to the argument of the .DATA directive. All permanent labels defined in a .DATA section are of "full name" type, as explained in paragraph 3.4, i.e. they should be referenced by specifying (implicitely or explicitely) SegmentName.LabelName.

Example:

```
.DATA       MyDatSeg
```

NOTES:

1. Segment names are automatically PUBLIC and must be unique across all modules and all code and data segments, regardless of whether the segment is only used privately within a module.
2. Data segments may be split within a particular module, creating multiple sections of this segment. These sections are then glued together at link time to form one data segment according to the linking algorithm (see sections 4.4 and 5.5 for more details).

.DSECT SectionName

This directive defines the start of a new data segment. By default, the linker maps the .DSECT segments into the processor's data space regardless of whether it contains initialized data. The linker can be instructed, however, to map .DSECT segments into the processor's program (code) space, for example, to include constant tables or filter coefficients. When a new COFF section of type .DSECT is created, the previous temporary symbol table is deleted, and all permanent labels defined in the .DSECT section are of "simple" type, as explained in paragraph 3.3, i.e. they are referenced by specifying just their name. The .DSECT directive must be followed by a section name.

Example:

```
.DSECT      MyData
```

NOTES:

1. Segment names are automatically PUBLIC and must be unique across all modules and all code and data segments, regardless of whether the segment is only used privately within a module.
2. DSECT data sections may be split within a particular module, creating multiple sections of a segment. These sections are then glued together at link time to form one data segment according to the linking algorithm (see sections 4.4 and 5.5 for more details).

DW DataValue [,DataValue [,DataValue...]] or
DW NumericExpression DUP DataValue

This directive allocates one or more words of data which may be initialized. The DW directive is used like an instruction, in the sense that it is the only directive that is not prefixed by a dot (.). The DW directive must contain a list of one or more data values. Data values may be a numeric expression, a symbolic expression or uninitialized. Uninitialized values are signified by a '?'. Internally, uninitialized values are stored as zeros. The DW directive may only be used in a data segment and, like instructions, may be preceded by a label. The DUP operator may be used to repeat the initialization value, e.g.

Examples:

```
        DW    ?
        DW    1,2,3
        DW    3 DUP ?    ; translates to ?,?,?
        DW    2 DUP 5    ; translates to 5,5
        DW    "abcd"     ; translates to 0x6162,0x6364
%TempLabel:
        DW    1
PermLabel:
        DW    $
        DW    PermLabel
        DW    <%TempLabel
```

.EXTERN Symbol1 [,Symbol2 [,Symbol3...]]

This directive allows the use of symbols defined in another (external) assembly module (file). These symbols are resolved by the linker. Symbol names should be fully specified, i.e., prefixed with the appropriate segment name, unless the USE directive is used. To declare a segment name external without specifying any offsets, use Segment.0. Using symbols from external modules that have not been declared with either the EXTERN or GLOBAL directives will generate an assembly error during the relocation pass of the assembler.

Examples:

```
.EXTERN     Seg1.Offset2, Seg.Offset3
.USE        Seg2
.EXTERN     Offset4, Offset5
.EXTERN     Seg2.0
```

.FF

This is the form feed directive. It causes the start of a new page when the listing is printed.

.FORMAT LinesPerPage [,CharactersPerLine]

This is the format directive. It causes the listing to be formatted according to the specified values. By default, 66 lines per page are created and 80 characters per line.

.GLOBAL Symbol1 [,Symbol2 [,Symbol3...]]

The GLOBAL directive combines the EXTERN and PUBLIC directives, i.e., it can be used to specify symbols that will be imported from external modules and/or symbols that will be exported to other modules. It is useful for header files, since the same directive can be used for both the importing and exporting module. The disadvantage of this directive is that it could lead to multiple definitions of the same symbol in more than one module, which would cause an unresolved linker error.

Example:

```
.GLOBAL     Seg1.Offset1, Seg1.Offset2, Seg2.Offset3
```

.LIST BooleanNumericExpression

The LIST directive is used to switch source listing generation on and off. The default is 1 (on).

.ORG NumericExpression or
.ORG $+NumericExpression

The ORG directive sets the location counter of the current segment to the value specified by the argument. The ORG directive creates a new COFF section with the same name and type as the current segment. Internally, a CODE or DATA directive is generated.

.POPSEG

The POPSEG directive sets the current USE segment with the value obtained by popping the top entry from a segment name stack. In conjunction with the PUSHSEG directive, this directive is useful for writing nested macros that, when finished, will not have any effect on the current USE segment.

.PUBLIC Symbol1 [,Symbol2 [,Symbol3...]]

The PUBLIC directive is used to declare the symbols in its argument list (to be defined later in the module), as being exportable to other modules. These symbols can be declared in other modules with either the EXTERN or GLOBAL directives, and the linker will be able to resolve them. Like the EXTERN and GLOBAL directives, unqualified symbol names, i.e., symbol names not prefixed by a segment name, will use the current USE segment name by default.

.PUSHSEG

The PUSHSEG directive pushes the current USE segment name onto the top of the segment name stack. In conjunction with the POPSEG directive, this directive is very useful for writing nested macros that, when finished, will have no effect on the current USE segment.

.TITLE "text"

This is the title directive. Each new page following this directive, will have the title printed on the top of page, below the fixed header (company logo and version number), the date and time of printing and the page number.

.USE [SegmentName]

The USE directive specifies the current USE segment name to be used when encountering an unqualified permanent symbol reference. When no argument is specified, the name of the current segment is used. The current USE segment name is affected by the following directives: CODE, DATA, ORG, and POPSEG.

3.8 Macro Preprocessor Directives

In addition to the assembler directives, the following are supported by the macro preprocessor:

.INCLUDE "FileName"

The INCLUDE directive is identical to the C "#include" directive, i.e., it instructs the assembler to read and merge another module (file) into the source file at the line where this directive is located. Conventional completely specified DOS path names of files may be used to access files outside the working directory. Alternatively, an environment variable, named MPP, may be set (using the DOS command SET) to tell the macro preprocessor to look for the file in the path specified by the MPP variable in case the file is not found in the current (working) directory. Multiple directories may be specified by seperating with a semicolon (;). In addition, one can overwrite the environment variable path, by using the -i or -I option when invoking the macro preprocessor (see also Section 3.2.4).

NOTES:

1. One can nest included files up to 14 levels.
2. Up to 10 paths may be specified and each is limited to 80 characters.

3.8.1 Conditional Directives

.IF BooleanExpression

The IF directive marks the beginning of a conditional block. It is used to control the assembly conditionally as follows: If the boolean expression is evaluated as true, then the source lines in the conditional block (following the IF directive up to another conditional directive) are included in the source. If the boolean expression is false, the conditional block is ignored.

.IFDEF Symbol

The IFDEF directive marks the beginning of a conditional block. It is used to control the assembly conditionally as follows: If the symbol is defined previous to the current segment location counter, then the source lines in the conditional block are included in the source. If the symbol is undefined, the conditional block is ignored.

.IFNDEF Symbol

The IFNDEF directive marks the beginning of a conditional block. It is used to control the assembly conditionally as follows: If the symbol is not defined previous to the current segment location counter, then the source lines in the conditional block are included in the source. If the symbol is defined, the conditional block is ignored.

.ELSE

The ELSE directive marks the end of a previous conditional block and the beginning of a new conditional block. It is used to control the assembly conditionally as follows: If the boolean expression associated with the previous conditional block is evaluated as false, then the new conditional block (following the ELSE directive) will be included in the source.

.ELIF BooleanExpression

The ELIF directive marks the beginning of a nested conditional block. It is used to control the assembly conditionally as follows: If the boolean expression is evaluated as true, then the next conditional block is included in the source. If the boolean expression is false, then the conditional block is ignored.

.ENDIF

The ENDIF directive marks the end of a conditional block. (See also previous conditional directives).

Examples of conditional directives:

```
.IF Flag>0
     mov #0x0f, r1
.ELSE
     mov #0,r2
.ENDIF

.IFDEF seg1.Offset1
     clra0
.ELIF Flag<2*4
     mov #1,a11
.ELSE
     mov #2,a11
.ENDIF
```

3.8.2 Macro Directives

.ENDM

The ENDM directive terminates the definition of a new macro. A macro definition can not be nested inside another macro definition. See .MACRO for more details and examples.

.EQU Symbol FreeText or
.EQU Symbol(ParameterList) FreeText

The EQU directive is equivalent to the C "#define" directive, i.e., the macro preprocessor treats equates as literals (which may be nested). The first format is used for simple definitions of constants represented by symbols, the second format is used for smart C-like macros.

NOTES:

1. Equates to the PC location operator (.EQU var $) do not make sense and should not be used.
2. Equate expansion is delayed until the final stage, so nesting is possible.
3. Up to 3000 equates may be defined in one module.
4. The maximum length of each equate body is limited to 1000 characters.
5. The maximum number of parameters is 20.
6. In the second format, the opening parenthesis must immediately follow the symbol name.
7. Equates may be redefined. A warning message will be generated by the preprocessor.
8. If the equated symbol is not followed by free text, a default value of 1 is assumed. This can be used to define symbols for use by the conditional directives, without giving a value to the symbol. The .PURGE directive can then be used to undefine this symbol.

Examples:

```
.EQU  AAA   BBB
.EQU  BBB   111
      mov   #AAA, r0    ; translates to  mov #111, r0
.EQU  BBB   22          ; redefinition
      mov   #AAA,r1     ; translates to  mov #22, r1


.EQU  min(a,b)    ( ((a) <= (b))  ? (a) : (b) )
      add ##min(3,4), a0 ; translates to   add ##( ((3) <= (4) )
                           ;? (3) : (4) ), a0


.EQUFlag                ; equivalent to .EQU  Flag  1
.IFDEF Flag
   .INCLUDE "file"      ; file will be included
.ENDIF
.PURGE Flag
```

.MACRO Symbol [ParameterList]

The MACRO directive starts the definition of a new macro. A macro is composed of a declaration, a macro ending statement (see .ENDM) and a macro body of one or more assembly instructions in between. When the macro is declared, it is given a unique name and an optional parameter list. As in C, macros are treated as literals. Unlike C, the parameter list is defined or referenced without surrounding parenthesis. The main difference between a macro and an equate is that macros can expand over multiple lines. Macro definitions may not include other macro definitions, but macros can use previously defined macros so nesting is possible.

NOTES and LIMITATIONS:

1. A maximum of 1000 macros is allowed in each module.
2. Macros may not contain more than 1200 characters inside the body of the macro.
3. A maximum of 20parameters is allowed. Each parameter is limited to 200 characters.
4. Macro names are not allowed with embedded white space characters.
5. Macro definitions appear in the listing file as a comment. The expansion of macros in the listing is controllable by a switch. An additional ".X" directive is generated in the listing by the macro preprocessor for each macro that has more than one line in its body, for the purpose of synchronizing reports on erroneous lines by DM_ASM.
6. Macro definitions can not be nested. However a macro definition can use a previously defined macro (forward references are not allowed).
7. Single and multiple line comments (/* comment */) can not be used inside macros.

Macro definition examples:

```
.MACRO MyMac
     mov   #0, r0
     mov   #1, r1
.ENDM

.MACRO MyMacWithArgsN0,N1,N2
     mov   #N0, r0
     mov   #N1, r1
     mov   #N2, r2
.ENDM


.MACRO NestedMac
     clr   a0
     MyMac
.ENDM
```

Macro usage examples:

```
MyMac                    /* mov #0, r0
                            mov #1, r1 */

MyMacWithArgs  4,5,6     /* mov #4, r0
                            mov #5, r1
                            mov #6, r2 */

NestedMac                /* clr  a0
                            mov #0, r0
                            mov #1, r1 */
```

.PURGE Symbol

The PURGE directive is the equivalent of the C "#undef" directive. It ends the definition of a symbol.

Example:

```
.PURGE   Seg1.Label
```

3.9 DM_ASM Limitations

The assembler has the following size limitations:

1. A module can not contain more than 3072 symbols. Symbols include all segment names, offset names and the segment name "CODE" which is always defined by default.
2. A module can not contain more than 63 COFF sections. Every use of the directives .DATA, .CODE and .ORG creates a new COFF section regardless of whether the segment has been previously defined.
3. A COFF section may not contain more than 64K of code or data. Note that data segments larger than 256 words can not take advantage of the processor's directaddressing mode. The linker will check that the 64K limits are not exceeded after combining all sections.
4. A single COFF section may not contain more than a combination of 256 temporary label definitions and references.
5. Segment names may not exceed 8 characters and must be globally unique.
6. Offset names may not exceed 31 characters and must be unique within their segment.
7. Temporary labels may not exceed 31 characters and need not be unique (but their scope is only for the current segment).
8. The maximum size of the segment name stack activated by .PUSHSEG and .POPSEG directives is 16 entries.
9. The .EXTERN, .PUBLIC, and .GLOBAL assembler directives can accept a list of up to 10 label names. There is no direct limit to the number of times that the directives may be repeated (with other label names).

SECTION 4
DM_COFFLink Description

4.1 General Notes

The CoffLink COFF linker is designed to link COFF object modules created by the PineASM COFF assembler version 6.x or higher or by the CoffLib COFF library archiver. The linker can be used to locate segments at absolute locations or relative to other segments (with an address alignment option), and to overlay segments. The assembler does not support any absolute location operators. All linking/locating must be specified via the linker. The Pine architecture, assembler and linker work in harmony with each other. For example, in order to accommodate the processor's direct addressing mode, the Pine has the **lpg** #immediate instruction, the assembler supports link-time @ and **PG** operators, and the linker supports the directives **align** and **inpage** that can all be used to support worry-free direct addressing. The linker supports overlays in both the code and data memory spaces, for efficient on-chip data memory usage and program downloading applications. The linker works with libraries. The linker supports user defined memory classes, and has by default, two address spaces/classes that are predefined and correspond to the processor's physical code and data memory spaces. The linker is an open system. Commands are entered via a script file referred to hereafter as the linker script file.

The linker is installed together with the assembler. See Section 2 for installation instructions.

4.2 DM_COFFLINK Invocation

There are two ways to invoke the linker. The usual way is to activate the linker via a batch file named LINK6.BAT, the second way is to run the main program, COFFLNK6.EXE, directly from the DOS command line.

4.2.1 Batch File Invocation

When running the LINK6.BAT batch file, the arguments are the various optional files and the base name of the linker script file without the mandatory .LNK extension, e.g.

```
LINK6 [options] LinkerScriptBaseFileName
```

The output from LINK6.BAT are a COFF executable file (.A) and a listing file (.LIN), each of which can be obtained independently of the other with optional user-supplied filename.

The options are:

`-h`	help information
`-p`	invoke the MPP before the linker
`-m`	create section map
`-s`	create symbol table
`-x`	create cross reference index

4.2.2 Command Line Invocation

The linker executable, COFFLNK6.EXE, can be invoked directly from the DOS command line as follows:

```
COFFLNK6 [option] < LinkerScriptFileName
```

where the options are as follows:

`-h`	help information
`-S`	script file is taken from the standard input
`-L`	output is directed to the standard output
`-l file`	force output file name for linker listing (full name is required)
`-o file`	force output file name for COFF executable (full name is required)
`-p`	invoke the MPP before the linker
`-m`	create section map
`-s`	create symbol table
`-x`	create cross reference index
`-w`	wrap mode active in listing file

4.3 DM_COFFLINK Script File

The linker script file has several interleavable sections (parts): **classes, objects, libraries, code,** and **data.** Multiple instances of each section are allowed. Only the **objects** section is mandatory. Section order relative to the other sections is usually not important, but the order of the contents within each section (after combining multiple instances) is very important - it dictates the order of the link algorithm. Code and data segments not explicitly mentioned in the script file, are linked according to a default algorithm. The complete linking algorithm is detailed in Section 4.4. The **classes** section is used to define a list of memory types, each associated with a memory range. The default classes are **code** and **data**, each having a memory range of 0x0000-0xffff. The **objects** section contains an ordered list of all object files to be linked. The **libraries** script section contains an ordered list of all library objects to be linked. The section labels, i.e.ä **classes, objects, libraries, code** or **data**, must be on a line by themselves followed immediately by a colon (:). Only one file/ segment per line is permitted. Lines may contain a trailing comment signified by a semi-colon (;). Blank lines are permitted. The script file is case sensitive except for file names. The following figure describes the general structure of the linker script file.

Structure of the linker script file:

```
objects:
     ListOfObjectFiles
```

```
libraries:
     ListOfLibraryFiles

classes:
     ListOfClassDeclarations

class1:
     ListOfSegmentsWithOptionalAttributes

class2:
     ListOfSegmentsWithOptionalAttributes

...

classN:
     ListOfSegmentsWithOptionalAttributes
```

Only one file or segment per line is permitted. Lines may contain a trailing comment signified by a semi-colon (;). Blank lines are permitted. The script file is case sensitive except for file names. The following are two typical examples of a linker script files. The first is for linking together 3 object files, declaring two user defined memory classes (in addition to the two predefined classes) and for locating particular segments at specified addresses.

Example 1:

```
objects:
     file1.o
     c:\mypath\subdirectory\file2.o
     ..\..\file3.o

classes:
     xram [d:0000,d:03ff]
     yram [d:fc00,d:ffff]

code:
     Segment1                    ; will be located at 0x0000
     Segment2  at    0x8         ; will be located at 0x0008
     Segment3

data:
     Segment4  at    0x8000 inpage
     Segment5  align 0x100  inpage ; will be after Segment4

xram:
     Segment6                      ; will be located at 0x0000
     Segment7  align 0x100  inpage ; should not exceed 0x03ff

   yram:
     Segment8                      ; will be located at 0xfc00
```

The second example uses libraries and creates a data overlay.

Example 2:

```
objects:
   file1.o
   c:\mypath\file2.o
   ..\file3.o

libraries:
   file4.lib
   ..\lib\file5.lib

code:
   Segment1                    ; will be located at 0x0000
   Segment2  at    0x100       ; will be located at 0x0100
   Segment3  lo                ; linker will start trying
                               ; to locate from address 0

data:
   Segment4
   Segment5  inpage            ; will be after Segment4
     {
        Segment6               ; located after Segment5
        Segment7  align 0x100  inpage
        Segment8  at  Segment6
     }
```

The linker script file has several interleavable sections (parts): **classes**, **objects**, **code**, and **data**. Multiple instances of each section are allowed. Only the **objects** section is mandatory. Section order relative to the other sections is usually not important, but the order of the contents within each section (after combining multiple instances) is very important - it dictates the order of the link algorithm. Code and data segments not explicitely mentioned in the script file, are linked according to a default algorithm. The complete linking algorithm is detailed in paragraph 4.4. The **classes** section is used to define a list of memory types, each associated with a memory range. The default classes are **code** and **data**, each having a memory range of 0x0000-0xffff. The **objects** section contains an ordered list of all object files to be linked. The section labels, i.e. **classes**, **objects**, **code** or **data**, must be on a line by themselves followed immediately by a colon (:). Only one file/segment per line is permitted. Lines may contain a trailing comment signified by a semi-colon (;). Blank lines are permitted. The script file is case sensitive except for file names.

4.3.1 Linker Directives

The complete list of reserved linker directives is:

```
at          libraries
align       lo
classes:    next
code:       next (...)
data:       noload
hi          objects:
impage
```

The optional location attributes **at, align, image, hi, lo,** and **next** allow the user to specify how he would like the linker to locate the various data and code segments (included in the object files mentioned in the **objects** section). Each memory class defined in the **classes** section (including the default **code** and **data** classes) is described separately in a different section of the script file. The **emulator** attribute is added to those segments that belong to PICEOS.

To locate a segment, one and only one of the following location attributes must be explicitly or implicitly associated with each segment:

at ConstantNumericExpression
at SegmentName
at hi
at lo
at next
at next(AddressExpressionList)
lo
next
next(AddressExpressionList)

An AddressExpressionList is a list of symbolic addresses, e.g.

```
(SegA,SegB,SegC)
(SegA+10,SegB-5)
(100)
(next,lo+100)
(hi-0x100)
```

All the above location attribute combinations can be appended with a numeric offset or the align attribute, e.g.

```
at SegA + 7  align  0x100
next(SegB+6,SegC) - 10
at hi - 0x200
at next(0x100)          ; equivalent to 'at 0x100'
at 0x100  noload
```

Notice that the linker will surround each subexpression after each + or - operator, by parenthesis. This means that

```
at next (segA - 0xff +4)
```

is actually interpreted as

```
at next(segA -(0xff +4))
```

or

```
at (SegB -100 - 0x15 + 3)
```

as

```
at (SegB - (100 - (0x15 + 3)))
```

align

The **align** directive is followed by a numeric constant value. It is used to force the linker to place the relevant segment at an address that is an integer multiple of the numeric constant. It can be appended to any other location expression.

Examples:

```
SegA
SegB  align 0x100 ; SegB after SegA on address that is multiple of 0x100
SegC  at  SegB+50 align 4 ; SegC at SegB+50, but align to a multiple of 4
```

at

The **at** directive must be followed by an address expression. An address expression can be a numeric value, a simple numeric expression, the linker directive **lo** or **hi**, a reference to a previously located segment (no forward references are allowed) or a reference to a **next** expression. When the **at** attribute is used in the linker script file, it means that the linker must locate the associated segment at the indicated address expression, subject to an optional alignment.

Example:

```
SegA  at 0x500      ; locate SegA at address 0x500
SegB  at SegA+0x100 ; locate SegB at address 0x600
SegC  at next       ; put SegC at the next available address after SegB
```

hi

The **hi** directive stands for the highest address of the memory class. For the predefined **code** and **data** classes, **hi**=0xffff. It must be used together with a negative offset after an **at** directive.

Example:

```
classes:
    yram [d:fc00,d:ffff]
yram:
    Seg1 at  hi - 0x200    ; locate Seg1 at 0xfe00
    Seg2                   ; locate Seg2 after Seg1
```

inpage

The **inpage** directive can be used with data segments to request the linker to check that the associated segment does not cross physical page boundaries (each 256 data words long), and issue a warning if the segment does.

Examples:

```
data:
    Seg1   inpage       ; put Seg1 at 0x0000, make sure it is shorter than 256
    Seg2   at    0x180  ; locate Seg2 at 0x180
    Seg3   inpage       ; put Seg3 after Seg2, make sure it does not cross 0x200
```

lo

The **lo** directive stands for the lowest address of the memory class. For the predefined **code** and **data** classes, **lo**=0x0000. When the **lo** attribute is used in the linker script file, it means that the linker must start searching from the lowest address of the memory class, subject to an optional alignment constraint. By default, if no location attribute is assigned to the first segment in the class, it is located at the first address of the class, i.e. at lo.

Example:

```
code:
    Seg0   at   0x100     ; locate Seg0 at 0x100
    Seg1   at   lo        ; locate Seg1 at 0x0000
    Seg2   at   0x1000    ; locate Seg2 at address 0x1000
    Seg3   lo             ; try to locate Seg3 before Seg0 or Seg2 if possible
```

next and next(List)

The **next** directive is used to indicate the next available free hole's address that fits the segment. It can be followed by an address expression list in which case it means the next available address after all specified addresses. When the **next** attribute is used (or when no attributes are given at all), it instructs the linker to start searching from the address immediately following the hi address of the previous segment that was located, subject to an optional alignment constraint. When

next(List) is used, the effect is similar to the simple next case described above, except that the search begins with the address following the maximal hi address of all segments in the list. When the list contains a constant numeric expression, the hi address is one less than the value of the expression (so that the search can start at the value).

Example:

```
data:
        Seg1    at      0x1000          ; locate Seg1 at address 0x1000
        {
          Seg2                          ; locate Seg2 after Seg1
          Seg3  at  Seg2                ; locate Seg3 at the beginning of Seg2
          Seg4  at  next(Seg2,Seg3)     ; locate Seg4 after the longer section
                                        ; among Seg2 and Seg3
          Seg5  next                    ; locate Seg5 after end of Seg4
        }
```

noload

The optional **noload** directive is used for segments that are linked with other segments, but that should not be loaded by the loader of the debugger at load time.

4.3.2 Libraries Script Section

The **libraries** script section contains an ordered list of all library object files to be linked.

The following is an example of a libaries script section:

```
libraries:
        lib1.lib
        \path\lib2.lib
        lib3.xyz
        mylib.          ; mylib. (note that no default extension is appended)
```

4.3.3 Objects Section

The **objects** section contains an ordered list of all object files to be linked. File names without extensions will be considered as having a default extension of ".o".

The following is an example of an objects section:

objects:

```
mod1        ; mod1.o
mod2.o      ; mod2.o
\path\mod3  ; \path\mod3.o
mod4.xyz    ; mod4.xyz
mod5.       ; mod5. (note that no default extension is appended)
```

4.3.4 Classes Section

The **classes** script section defines a list of logical memory types of the target executable COFF file. Up to 14 classes may be defined by the user. Each class is assigned a memory range (defining the **lo** and **hi** addresses of the class) in either the program (code) space or the data space. For each memory class defined, the programmer should add a script section in the linker script file to specify which segments (.CODE, .CSECT, .DATA and .DSECT sections) declared in the object files, belong to that memory class. During the linking process the linker makes sure that the appropriate segments fit into these ranges. The **classes** script section is entirely optional. By default, two memory classes are predefined, the **code** and **data** memory classes, each having the default range of the entire program (code) and data memory spaces respectively, e.g. they have the range of 0x0000-0xffff. Memory classes may overlap in their address ranges, however a segment can not implicitely overlap in two different classes. This means that once a particular segment in a particular class, is located in either the program (code) or data memory space, another segment, in the same or in another class, can not be mapped into the same memory addresses (in the same memory space) occupied by the first segment, unless an overlay group is explicitely declared, as described below. The following is an example of a typical **classes** script section:

The following is an example of a typical **classes** section:

classes:

```
xram    [d:0000,d:03ff]  ; user defined class for on-chip xram
yram    [d:fc00,d:ffff]  ; user defined class for on-chip yram
eprom   [c:8000,c:bfff]  ; user defined class for external eprom
```

Note that by default the following two classes are always predefined:

```
code    [c:0000,c:ffff]    ; defined by default
data    [d:0000,d:ffff]    ; defined by default
```

Once classes are defined, one should specify for each class which segment belong to the class. In the example above, one can add 5 script sections, named **code:**, **data:**, **xram:**, **yram:** and **eprom:**. Using linker attributes, the programmer can instruct the linker to locate some or all the segments into specific memory locations. See the linker location attributes description above and more details in paragraphs 4.3.4 and 4.3.5 describing the **code** and **data** script sections. Segments defined in the object files, that are not explicitely mentioned in any of the class script sections, are mapped as described by the default linking algorithm, i.e. **.CODE** and **.CSECT** sections are mapped into the default **code:** script section (after the last segment already mapped), and **.DATA** and **.DSECT** sections are mapped into the default **data:** script section, (after the last segment already mapped). Classes should be used to guarantee that particular data structures or programs fit into physical memory devices or memory limits imposed by a particular chip configuration.

4.3.5 Code Section

The **code** script section specifies which segments (.CODE, .CSECT, .DATA and .DSECT sections) defined in the object files mentioned in the objects: script section, are to be linked in which order into Pine's default program (code) memory class. The syntax for specifying a segment is as follows:

```
SegmentName [at [hi | lo | next[(list)]] SymbolicNumExpr] [align NumExpr] [noload]
```

or

```
SegmentName [lo | next] [ align NumExpr] [nolad]
```

All attributes are optional. All can be appended with a +/- constant numeric expression offset (e.g. **at** SegA + 0x0f1) or with the directives **align** or **noload**. The **at** attribute specifies an exact address in which to map the segment. The **align** attribute specifies that the segment must be mapped on the next address, which is a multiple of the specified numeric expression. The **lo** attribute is used to instruct the linker to map starting from the memory class' lowest address (default 0 for **code** class), even if other segments have already been mapped at higher addresses, obviously, without causing overlapping of segments. The **hi** attribute can be used to instruct thelinker to map and fill the memory space relative to the memory class' highest address, (i.e. address 0xffff in the **code** class). The **next** attribute specifies that the segment must be mapped immediately after the previous segment. The segments in the **code** class are not necessarily only .CODE and .CSECT sections. ROM tables, for example, are .DATA or .DSECT sections that can also be linked into the program (code) space. The SymbolicNumExpression is a C-style numeric expression that may contain one or more segment names that have already been located (no forward references are allowed).

The following is an example of a typical code script section:

```
code:
      seg1
      seg2     align  0x100
      seg3     lo
      seg4     at     seg2 + 0x600  noload
      romtbl   next
```

To support program downloading, code segments may overlap other code segments if specified so using overlay groups. This is useful when a relatively small program RAM is available which is used to run different applications or program sections downloaded from slow EPROMs in the data space, at different run times. To create downloadable programs, a particular segment may be linked twice: once in the program space (where it is down-loaded to and executed) and once in the data space (where it loaded from). See paragraph 5.x for further details and examples.

Code Overlays

Segments within the code script section may be overlaid to allow multiple views of the same program address space. To overlay code segments, these segments must belong to an overlay group, which can be viewed as one logical segment. Overlay groups are created by surrounding the member segments with braces { }. An overlay group is restricted to fit inside its class boundaries (just as any normal segment). Multiple overlay groups are allowed per class, but overlay groups may not be nested or overlap each other. The **next** address following an overlay group is the maximum **next**

address of all the member segments. In terms of syntax, data and code overlays are identical. Note that to facilitate downloading, it is possible to use the **SIZEOF** and **INCODE** operators to obtain the length of a section and to refer to the address in the program space of a symbol that is linked into both the program and data memory spaces. See the description of the assembler operators and directives for more details. See section 5.x for a typical downloadable application. The following example defines a code overlay:

```
code:
    Reset      at 0x0000
    {
        Prog1
        Prog2 at Prog1
    }
```

See data overlays for further details and limitations concerning overlay groups.

4.3.6 Data Section

The data section specifies which data segments are to be linked in which order into Pine's default data memory space. The syntax for specifying a segment (except within overlays) is as follows:

SegmentName [**at** [**hi** | **lo** | **next**[(list)]] SymbolicNumExpr] [**align**NumExpr] [**inpage**] [**noload**]

or

SegmentName [**lo** | **next**] [**align**NumericExpr] [**inpage**] [**noload**]

All attributes are optional. All can be appended with a +/- constant numeric expression offset (e.g. **at** SegA + 0x0f1), the **align, inpage** or **noload** attributes. The **at** attribute specifies an exact address in which to map the segment. The **align** attribute specifies that the segment must be mapped on the next address of the specified multiple. The **lo** attribute is used to instruct the linker to map starting from the memory class' lowest address (default 0 for **data** class), even if other segments have already been mapped at higher addresses, obviously, without causing overlapping of segments. The **hi** attribute can be used to instruct the linker to map relative to the memory class' highest address (default 0xffff for the **data** class). The **next** attribute specifies that the segment must be mapped immediately after the previous segment. The **inpage** attribute must be used for data segments that must not cross the physical page boundaries in the data space. The **noload** attribute will cause the loader of the debugger not to load that segment at load time.

The following is an example of a typical data section:

```
data:
    seg1
    seg2   at       0x240
    seg3   align    0x100  inpage
    seg4   inpage   noload
```

Usually the data memory space on the Pine chip is limited, so the programmer is forced to use the same address space for different data segments. This can be accomplished using data overlays.

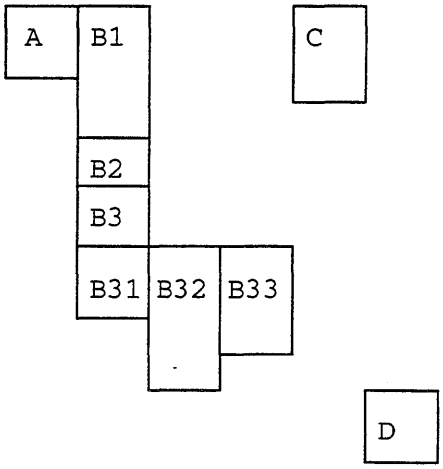
4.3.7 Data Overlays

Data segments within data sections may be overlaid to allow multiple views of the same data address space, i.e. C-style unions. To overlay data segments, these segments must belong to an overlay group, which can be viewed as a logical segment. Overlay groups are created by surrounding the member segments with braces { }. An overlay group is restricted to fit inside its class boundaries (just as any normal segment). Multiple overlay groups are allowed per class, but overlay groups may not be nested. The 'next' address following an overlay group is the maximum 'next' address of all the member segments.

The following is an example of a data overlay group defined in memory class MyClass. On the right is a picture reflecting the memory map defined by this overlay group.

MyClass:

```
{     ; begin overlay group
  SegA
  SegB1      at  SegA
  SegB2
  SegB3
  SegB31
  SegB32     at  SegB31
  SegB33     at  SegB31
  SegC       at  SegA

}     ; end overlay group

SegD
```



NOTES and LIMITATIONS:

1. Segments defined in an overlay group may overlay only onto other segments in the same group. Segments defined in an overlay group may not overlay onto segments outside the group.
2. No attributes may be assigned to the overlay group as a whole, i.e. on the line containing the open brace {. Attributes can only be assigned to the individual segments that are members of the group.
3. The default location attribute given to segments within the group is **at next**.
4. Holes created inside an overlay group (as a result of using the **at** attribute) can not be filled by sections outside the overlay group.

4.4 Linking Algorithm

The linking algorithm for the code and data sections is as follows:

1. The program space segments (i.e. .CODE, .CSECT, .DATA and .DSECT sections that are explicitly specified in the code script section) are mapped in the order in which they are

encountered. If a segment has no associated location attributes, it is mapped immediately after the end of the previous segment. If the **at** attribute is used, the segment will be mapped starting at the specified address. If the **align** attribute is used, the segment will be mapped at the next address on the specified alignment boundary. If the **next** attribute is used, it is mapped immediately after the end of the previous segment. If the **lo** attribute is used, mapping starts from the lowest address upwards. Segments can be grouped together to form an overlay group. The segment group is treated as if it is a logical segment. Its length is taken as the difference between the lowest and highest addresses of all the segments in the group. A segment following an overlay group will be located immediately after the highest address occupied by any of the member segments.

2. Remaining .CODE and .CSECT sections that have not yet been mapped (i.e. those that are not explicitely mentioned in the script file) are mapped immediately after the last segment mentioned and mapped into the program space, into the **code** class, according to rule 1 above. If no segments were specified in the code script section, then mapping begins at the lowest address. The remaining .CODE and .CSECT sections are mapped in the order of the first time they are encountered in the (ordered) list of object files found in the object section.

3. The data space segments (i.e. .DATA, .DSECT, .CODE and .CSECT sections that are explicitly specified in the data script section) are mapped in the order they are encountered. If a segment has no associated location attributes, it is mapped immediately after the end of the previous segment. If the **at** attribute is used, the segment will be mapped beginning at the specified address. If the **align** attribute is used, the segment will be mapped at the next address on the specified alignment boundary. If the **inpage** attribute is used, the linker will check that the segment does not cross a physical page boundary (any address of type 0xYY00). If the **next** attribute is used, it is mapped immediately after the end of the previous segment. If the **lo** attribute is used, mapping starts from the lowest address upwards. Segments can be grouped together to form an overlay group. The segment group is treated as if it is a logical segment. Its length is taken as the difference between the lowest and highest addresses of all the segments in the group. A segment following an overlay group will be located immediately after the highest address occupied by any of the member segments.

4. Remaining .DATA and .DSECT sections that have not yet been mapped (i.e. those that are not explicitely mentioned in the script file) are mapped immediately after the last segment mentioned and mapped into the data space, in the **data** class, according to rule 3 above. If no segments were specified in the data script section, then mapping beginsat the lowest address. The remaining .DAT and .DSECT sections are mapped in the order of the first time they are encountered in the (ordered) list of object files.

5. Data and code overlays are allowed only when explicitly specified. Implicit overlays resulting from the use of the **at** or **align** directives will generate an error.

6. The linker issues a warning when it detects relocation size conflicts, i.e., the size of a symbol's address after having been fully resolved requires more bits than available in the operand field of the instruction.

7. The linker issues a warning when it detects a data segment, associated with the **inpage** attribute, that crosses a physical page boundary (every 256 words is a defined as a physical page, so every address 0xYY00 is considered such a page boundary).

8. The linker will mark segments associated with the **noload** attribute, so that the loader of the debugger will not load that segment into memory at load time.

4.5 Generating COFF Library Files

It is possible to create COFF libraries from one or more simple COFF object files using the COFFLIB utility. This utility converts COFF object files (.O) into COFF library files (.LIB) using the following command:

```
COFFLIB  command  LibraryName
```

where the command is one of the following:

`-h`	help information
`-a ObjectFile`	add the specified module to the COFF library
`-x ObjectFile`	extract the specified module from the COFF library
`-v`	view the COFF library contents

For example, suppose you have 3 files, each containing two library functions:

file1.asm	produces	file1.o	contains	func1a and func1b
file2.asm	produces	file2.o	contains	func2a and func2b
file3.asm	produces	file3.o	contains	func3a and func3b

Then the following commands create a COFF library named MYLIB.LIB with these functions:

```
COFFLIB  -a file1.o MYLIB.LIB
COFFLIB  -a file2.o MYLIB.LIB
COFFLIB  -a file3.o MYLIB.LIB
```

and the following command verifies the results:

```
COFFLIB  -v  MYLIB.LIB
```

which should return the following list:

```
file1.o   func1a  and  func1b
file2.o   func2a  and  func2b
file3.o   func3a  and  func3b
```

It is possible to go back from an archived library module to a simple COFF object file, using the extract command. For example to get back file2.o from the above library, enter:

```
COFFLIB  -x file2.o MYLIB.LIB
```

4.6 Generating PROM Burnable Files

Most EPROM programmers do not accept COFF object files as input. A utility can be used to convert COFF executable files (.A) into byte-wise Intel-Hex format files (.HCL,.HCH,.HDL,.HDH). The Intel-Hex file can also be used for loading programs into the debugger but without any symbolic data. The conversion can be activated in two ways: via batch file and directly from the DOS command line.

Using the batch file, the argument is the base name of a COFF executable file without the mandatory (.A) extension:

```
COFF2HEX CoffExecutableBaseFileName
```

The output from COFF2HEX.BAT are the four byte-wise INTEL-HEX files:

```
CoffExecutableBaseFileName.HCL
CoffExecutableBaseFileName.HCH
CoffExecutableBaseFileName.HDL
CoffExecutableBaseFileName.HDH
```

To convert directly from the DOS command line, use the program INTELHEX.COM. The input must be prepared in the appropriate format, i.e. a ordered list of hexadecimal addresses NNNN (with c: or d: prefix for indicating the memory space) followed by the hexadecimal value MMMM as described below:

```
C:NNNN  MMMM
C:NNNN  MMMM
...
D:NNNN  MMMM
D:NNNN  MMMM
...
```

The program COFFUTIL.EXE can be used to obtain this file from the binary COFF executable file. The syntax for invoking both programs from the DOS command line is as follows:

```
COFFUTIL -c CoffFile > DataFile
INTELHEX < DataFile
```

The output from INTELHEX.COM are the four byte-wise INTEL-HEX files shown above. Normally, there is no reason to directly invoke this program from the DOS command line.

4.7 DM_COFFLINK Limitations

1. There is a limit of 3K symbols.
2. There is a limit of 256 COFF sections.
3. There is a limit of 128 object files.
4. There is a limit of 128 segments.
5. There is a limit of 16 memory classes (of which 2, 'code' and 'data' are predefined).
6. There is a maximum of 16K cross reference entries.

4.8 DM_COFFLINK Error Messages

Link Errors:

"Illegal link relocation type at %s+0x%4.4X, symbol '%s'"
"Incorrect s_flag '%lx' in section header of object file '%s'"
"Multiple definitions of label '%s' in file '%s'"
"Object file '%s' contained warnings"
"Reference to '%s' is not resolvable as a base address for '%s'"
"Relocation size conflict at %s+0x%4.4X, symbol '%s'"
"Segment '%s' is used for both CODE and DATA"
"Undefined Extern symbol: %s"
"Undefined Global symbol (Slipped through??): %s"
"Undefined Public symbol: %s"
"Undefined symbol: %s"
"Unexpected segment overlay in segment '%s' address 0x%4.4X"
"Unspecified segment type for segment '%s'. Internal software error"

I/O Errors

"Cannot re-read relocation entries from output file"
"Cannot re-read section contents"
"Cannot re-write section '%s' contents"
"Cannot re-write section contents"
"Cannot read contents of object file 's'"
"Cannot read object file '%s'"
"Cannot read relocation info of file '%s'"
"Cannot read string table count from object file '%s'"
"Cannot read string table from object file '%s'"
"Cannot read symbol table info of file '%s'"
"Cannot write contents of file '%s'"
"Cannot write file header of output file"
"Cannot write section header '%s'"
"Cannot write section relocation info of file '%s'"
"Cannot write string table count"
"Cannot write string table of object file '%s'"
"Cannot write symbol table info of file '%s'"
"Unable to open object file '%s'"
"Unable to open output file"
"Unable to read file header of object file '%s'"
"Unable to read section header of object file '%s'"
"Unable to read string table of file '%s'"

Limitation Errors

"More than maximum allowed code segments (%d)"
"More than maximum allowed data segments (%d)"
"More than maximum allowed object files (%d)"
"Section '%s' exceeded 64k"
"Segment %s exceeds the maximum allowed segments (%d)"

Memory Allocation Errors

"HashTbl is full"
"Unable to allocate memory for bit maps"
"Unable to allocate memory for symbol manipulation"
"Unable to allocate space for object file name '%s'"
"Unable to allocate space for symbol tables"
"Unable to create symbol table"

Internal Errors

"Unable to retrieve info from Hash Table for symbol:"

Linker Script File Errors

"Cannot find segment '%s'"
"Invalid syntax in linker directive file %s(%d)"
"Missing 'code:'or 'data:' directive"
"Missing 'object:'or 'code:'or 'data:' directive"
"Nested overlays not supported"
"No object files defined"
"Object file '%s' already listed, ignoring additional entry"
"Segment name '%s' already listed, ignoring additional entry"
"Unrecognized switch '%s\n"

Information/Report Messages:

"\nNo errors in Link.\n"
"\nTotal of %d linker errors. No executable file created.\n"

5.1 Data Structures

One-level deep data structure type definition macros can be created as follows:

```
.MACRO MyStruct
   Member1: DW ?
   Member2: DW ?,?
   Member3: DW ?
.ENDM
```

The above macro can be used to define the same data structure in more than one segment.

```
.DATA MySeg1
   MyStruct  ; defines MySeg1.Member1, MySeg1.Member2, MySeg1.Member3
.DATA MySeg2
   MyStruct  ; defines MySeg2.Member1, MySeg2.Member2, MySeg2.Member3
```

A segment independent macro can be defined which operates on the data structure.

```
.MACRO OperateOnMyStruct  Segment
   mov #Segment.Member1, (r0)+
   mov #Segment.Member2, (r1)-
   mov #Segment.Member3, (r2)+s
.ENDM
```

The macro can be used as follows:

```
OperateOnMyStruct Seg1
OperateOnMyStruct Seg2
```

5.2 Safe Macros Using PUSHSEG and POPSEG

The following macro moves the specified label to the specified register. If the label is unqualified, i.e. it contains no segment prefix, the segment Seg1 is used as default. Even though the macro uses the USE directive, and therefore modifies the current USE segment, it is able to save and restore the state of the current USE segment of the caller (via the PUSHSEG and POPSEG directives).

```
.MACRO DefaultCopy  SrcLabel, TargetReg
   .PUSHSEG                  ; save USE segment
   .USE Seg1                 ; modify current USE segment
   mov ##SrcLabel,TargetReg
   .POPSEG                   ; restore USE segment
.ENDM

.CODE MyCodSeg
Label:
   nop
   DefaultCopy Member2, r0  ; mov ##Seg1.Member2, r0
   DefaultCopy Seg2.Member1,a0; mov ##Seg2.Member1, a0
   mov #Label, r1           ; mov #MyCodSeg.Label,r1
```

5.3 DIFF Equate

Normally, only one label is allowed in an operand expression. The IMMEDOFFSET operator can be used to convert the offset of a label (with respect to the segment in which it is defined) into an immediate numeric constant. Therefore any number of labels may be used in an operand expression as long as at least all but one are converted into constants by the IMMEDOFFSET operator. Recall that the IMMEDOFFSET operator can only handle labels that have been previously defined within the module, i.e. the labels cannot be external or forward references.

A common use of two labels in an operand expression is to calculate the difference, i.e. relative offset, between the location of the two labels:

```
.DATA MyDatSeg
...
LblA:    DW 5 DUP ?
...
LblB:    DW ?

.CODE MyCodSeg
...
mov #(IMMEDOFFSET MyDatSeg.LblB) - (IMMEDOFFSET MyDatSeg.LblA), r0
```

The code can be simplified with the following equate:

```
.EQU DIFF2(Label2,Label1)  ((IMMEDOFFSET Label2) - (IMMEDOFFSET Label1))
```

The equate can be used as follows:

```
mov #DIFF2(MyDatSeg.LblB,MyDatSeg.LblA), r0
```

Normally, it would be very poor programming practice to calculate the difference between two labels that were not defined in the same segment. In order to enforce this check, the DIFF equate could be modified as follows:

```
.EQU DIFF3(Seg,Label2,Label1)    DIFF2(Seg.Label2,Seg.Label1)
```

The protected equate can be used as follows:

```
mov #DIFF3(MyDatSeg,LblB,LblA), r0
```

The drawback of using the protected equate DIFF3 is that none of the labels can be temporary, since a temporary label can not contain a segment prefix. The unprotected equate, DIFF2, has no such drawback, since the (optional) segment prefix for each of the two label arguments must be explicitly supplied. Normally, it would be expected that the difference operator would be used for data structures defined with permanent labels. DIFF3 is preferred in more general cases.

5.4 Common Export/Import Include Files

Every assembly module should begin with a list of include files that define the module's exports and imports. The simplest way to organize a project is to break it down into its segments. Each segment should be assigned its own file set, e.g. MySeg.ASM and MySeg.INC, where the segment name and the file base name are identical. The .INC file should contain a USE directive followed by GLOBAL directives which enumerate all the public labels in the module. The .ASM file should start by including common macros, followed by including the module's export file, followed by including all of the module's import files, e.g.

```
;FILE: Seg1.ASM
; common project macros
.INCLUDE "PROJECT.MAC"
; module exports
.INCLUDE "Seg1.INC"
; module imports
.INCLUDE "Seg2.INC"
.INCLUDE "Seg3.INC"
; local equates and macros used by this module
; ...
; body of the module
.CODE Seg1
; ...

;FILE: Seg2.ASM
; common project macros
.INCLUDE "PROJECT.MAC"
; module exports
.INCLUDE "Seg2.INC"
; module imports
.INCLUDE "Seg3.INC"
.INCLUDE "Seg4.INC"
; local equates and macros used by this module
```

```
; ...
; body of the module
.CODE Seg2
; ...

;FILE: Seg1.INC
; equates and macros exported by this module
; ...
; labels exported by this module
.USE Seg1
.GLOBAL Label1, Label2, Label3

;FILE: Seg2.INC
; equates and macros exported by this module
; ...
; labels exported by this module
.USE Seg2
.GLOBAL Label1, Label2, Label3
```

In order to have a common header file for both exports and imports, when using the EXTERN and PUBLIC directives instead of the GLOBAL directive, the following trick could be used:

```
;FILE: Seg1.ASM
;
; common project macros
.INCLUDE "PROJECT.MAC"
;
; module exports
.EQU GLOBAL PUBLIC
.INCLUDE "Seg1.INC"
.PURGE GLOBAL
;
; module imports
.EQU GLOBAL EXTERN
.INCLUDE "Seg2.INC"
.INCLUDE "Seg3.INC"
;
; local equates and macros used by this module
; ...
; body of the module
.CODE Seg1
; ...
```

5.5 Multiple Segment Definitions

It is possible to define a segment in multiple parts in a single module. It is also possible to define a segment in multiple modules. The second practice is strongly discouraged. When defining A.A. segment multiply in a single module, the location counter of the segment continues from where it was previously, regardless of whether other segments have been defined in the interim.

Example:

```
.CODE Seg1   ; 1st instance of this segment - location counter is 0
...
.CODE Seg2
...
.DATA Seg3
...
.CODE Seg1   ; location counter automatically continues
```

The first time a segment is defined within a module, its location counter is initialized to zero.ä If a segment is defined in more than one module, then the ORG directive should be used in one or more of the modules in order to stop the segments from implicitly overlaying each other. The linker catches and warns when it detects this type of implicit overlay. Explicit overlays are allowed in data segments.

```
; FILE: MOD1.ASM
.CODE Seg1
...           ; location counter initialized to 0

; FILE: MOD2.ASM
.CODE Seg1
.ORG 0x80
...           ; location counter initialized to 0x80
```

5.6 Direct Memory Addressing Support

When using direct memory addressing in DMC, the opcode of the instruction supplies the lower 8 bits of the address, while the upper 8 bits of the address are supplied by the PAGE bits in status register ST1. Since it is very inconvenient to use absolute values (for the lower 8 bits of data addresses) inside the assembly program, one uses symbols defined in data segments. This way, when new symbols are added or old ones deleted or moved, the assembler and the linker take care of generating the correct lower 8 bits of the address. Each new data segment, starts a new series of consecutive symbols, starting from temporary address 0 upwards. At link time each such data segment can be located anywhere in the data space and the symbols corresponding to these segments will be relocated accordingly.

DM_ASM has an automatic modulo 256 operator for cutting the 8 lower bits of a 16 bit address (by the linker) for purposes of direct memory addressing. This is the @ operator. In addition, the OFFSET operator can be used, provided that all data segments are linked to be on DMC's page boundaries (i.e 0x0000, 0x0100, 0x0200, etc.) and are not longer than 256 addresses. The OFFSET operator tells the linker to put in the opcode of the instruction, the value corresponding to the offset of the symbol from the beginning of its data segment, i.e. it subtracts the absolute (final) 16 bit

address of the beginning of the segment in which the symbol is defined, from the absolute (final) 16 bit address of the symbol. As an example, suppose a program has the following code:

```
.DATA   SegA

VarA:   DW      ?
VarB:   DW      ?
VarC:   DW      ?

.CODE   SegB

    lpg     # SHR( SegA, 8)
    mov     OFFSET SegA.VarC, r1
```

and that the linker is instructed to locate segment SegA at 0x0100. The DM_ASM assembler will give a temporary address of 0 to SegA and SegA.VarA, a temporary address of 1 to SegA.VarB and temporary address 2 to SegA.VarC. Next, the DM_COFFLINK linker will give the final address of 0x100, 0x100, 0x101 and 0x102 to SegA, SegA.VarA, SegA.VarB and SegA.VarC respectively. In addition, in the code segment SegB, the linker will update the opcode for the **lpg** instruction by calculating the value of shifting SegA, i.e. 0x100, by 8 bits to the right. This results in putting the value 1 as the immediate operand of the first instruction. For the second instruction, the linker subtracts 0x100 (SegA) from 0x102 (the final value of SegA.VarC) so that the first operand of the **mov** instruction, (the direct memory address offset) will be 2.

What happens when there are many data segments ? If the data segments are not aligned on page boundaries, then the OFFSET operator will produce incorrect direct memory addresses. As an example, suppose that the same program is used as previously, but that the linker locates SegA at 0x205 (which is not aligned on a page boundary) as a result of another segment occupying the memory space up to address 0x204. In this case, the assembler will produce the same output, but the linker will give the final (absolute) addresses of 0x205, 0x205, 0x206 and 0x207 to SegA, SegA.VarA, SegA.VarB and SegA.VarC respectively. In addition, in the code segment, SegB, the linker will substitute the value 2 for the immediate operand of the **lpg** instruction (0x205 >>8 gives 2), and for the first operand of the **mov** instruction, it will still produce the value of 2, because SegA.VarC - SegA = 0x207 - 0x205 = 2. This however is not the correct direct memory address offset, needed to access SegA.VarC which has address 0x207, i.e. has direct memory offset of 7 in page 2.

The @ operator, on the other hand overcomes this problem, since it tells the linker to perform a modulo 256 operation on the absolute address, instead of the substraction operation. For example:

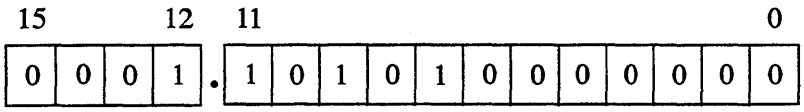
```
.DATA   SegA

VarA:   DW      ?
VarB:   DW      ?
VarC:   DW      ?

.CODE   SegB

    lpg     # SHR( SegA, 8)
    mov     @ SegA.VarC, r1
```

Whether the linker is instructed to locate the segment SegA at address 0x100 or 0x205, the object of the second instruction will be correctly "relocated", because 0x102 modulo 256 is 2 and 0x207 modulo 256 is 7. It is, therefore, better practice to use the @ operator for all direct memory addressing operands, and keep the OFFSET operator for use with data structures.

5.7 Fractional Arithmetics Support

Fractional arithmetics can be performed by a fixed point DSP such as the Drive Manager (DM) by allocating part of the 16 bits of a variable or memory location for the sign, the integer part and the fractional part. For example, let us define the Q_n binary fractional notation, where n bits are allocated for the fraction and 15-n bits are kept for the integer part of the number. The MS bit is used for the sign bit. In binary fractional numbers, each bit to the left of the floating point has the usual weight of 2^n, while the bits to the right of the floating point have a weight of $2^{-(n+1)}$.

As an example, consider the 16 bit number 0x3500 in Q_{12} notation.



The floating point value of this binary fractional number is: $2^1 + 2^0 + 2^{-2} + 2^{-4} = 3.3125$

The largest positive fraction in Q_{15} notation is very close to 1.0, represented by 0x7fff, while 0.5 is represented by 0x4000. In Q_{14} notation, on the other hand, 14 bits are used to represent the fractional part of the number, leaving 1 bit for the integer part. The largest positive number in Q_{14} notation is therefore 1.99999 (0x7fff). Note that Q_{15} fractions are more accurately represented than Q_{14} fractions, however numbers larger than 1.0 can not be represented by Q_{15} fractions. When multiplying binary fractional numbers, one needs to align the floating point in the result just as in decimal floating point arithmetics. This is accomplished by shifting the product one bit to the left and taking the high part of the shifted P register. For example assume one needs to calculate 0.5 * 0.5 = 0.25. Using Q_{15} notations, 0x4000 * 0x4000 = 0x10000000. The 32 bit result has Q_{30} notation (15+15 bits to the right of the floating point). By shifting the product 1 bit to the left, the high part of the product becomes 0x2000 which is 0.25 in Q_{15} notation.ä Shifting the product 1 bit to the left, corresponds to deleting 1 sign bit from the Q_{30} product that contains 2 sign bits (one from each multiplicand). As another example, suppose one multiplies a Q_{15} number by a Q_{12} number. The product will be a Q_{27} number, i.e. it has 27 bits representing the fractional part and 3 bits for the integer part.

The assembler has a built-in operator that automatically converts floating point numbers to 16-bit binary fractional numbers with a variable, user-specified, number of bits for the fractional part of the number.

Examples:

```
mov   ##FRACT(0.5, 15), x       ; translates to mov ##0x4000, x
mov   ##FRACT(0.015625, 15), r0 ; translates to mov ##0x200, r0
mov   ##FRACT(1.25, 14), y      ; translates to mov ##0x5000, y
```

For convenience, one can define a macro to simplify the notation as follows:

```
.EQU  Q15(num)    FRACT( num, 15 )
.EQU  Q14(num)    FRACT( num, 14 )
```

so that the previous examples can be rewritten as:

```
mov   ##Q15(0.5), x
mov   ##Q15(0.015625), r0
mov   ##Q14(1.25), y
```

with the same end result.

Note that the DSP architecture has a built-in shifter for the product register, that is specifically convenient for Q_{15} * Q_{15} operations. If all numbers are assumed to be in Q_{15} notation, the programmer can set the shift mode of the P register to be 1 bit to the left (SP=2 in ST1) and all results will be correctly aligned. By moving the product register to the accumulator and using combinations of the **shr**, **shl**, **shr4** and **shl4** instructions, it is possible to use all fractional notations to obtain fractional arithmetics with variable accuracy.

APPENDIX A
DSP Instruction Set

A.1 Notation and Conventions

Registers:

r_n = Address registers: r0, r1, r2, r3, r4, r5
r_i = Address registers: r0, r1, r2, r3
r_j = Address registers: r4, r5

aX = a0 or a1
aXl = Accumulator-low (LSP), X = 0, 1
aXh = Accumulator-high (MSP), X = 0, 1
aXe = Accumulator extension, X = 0, 1
ac = a0, a1, a0h, a1h, a0l, a1l

cfg_x = Configuration registers of DAAU (modi or modj, stepi or stepj), x = i, j

tos = Top of stack
pc = Program counter
lc = Loop counter
ext_x = External registers, x = 0, 1,...7

reg = a0, a1, a0h, a1h, a0l, a1l, r_n, x, y, p or ph, pc, lc, tos, st0, st1, st2, cfgi, cfgj, ext_x

Address Operands:

Address	= Unsigned 16 bits	(0 to 65535)
$Offset address	= 2's complement 7 bits	(-64 to 63 offset range: -63 to 64)

Immediate Operands:

#Short immediate	= Unsigned 8 bits	(0 to 255)
#Signed Short immediate	= 2's complement 8 bits	(-128 to 127)
##Long immediate	= 2's complement 16 bits	(-32,768 to +32,767)

cond - condition field:

true	Always
eq	Equal to zero
neq	Not equal to zero
gt	Greater than zero
ge	Greater or equal to zero
lt	Less than zero
le	Less or equal to zero
nn	Normalize flag is cleared
v	Overflow flag is set
c	Carry flag is set
e	Extension flag is set
l	Limit flag is set
nr	R flag is cleared
niu0	iuser0 input user pin 0 is cleared
iu0	iuser0 input user pin 0 is set
iu1	iuser1 input user pin 1 is set

Other:

(x)	= The contents of x
[ ]	= Optional field at the instruction
->	= Is assigned to
>>	= Shift right
<<	= Shift left
--	= Not
_	= Or
_	= And

Flags Notation:

The effect of each instruction on the flags is described by the following notation:

*	The flag is affected by the execution of the instruction.
-	The flag is not affected by the instruction.
1 or 0	The flag is unconditionally set or cleared by the instruction.

st0 bits	11	10	9	8	7	6	5	4
Flags	Z	M	N	V	C	E	L	R

Conventions

1. The arithmetic operations are performed in 2's complement.
2. When the $\mathbf{r_n}$ register is used by an instruction, the contents of the r_n register are post- modified as follows:

   Options controlled by instruction:

   r_n, r_n+1, r_n-1, r_n+s

   Options controlled by configuration registers cfg_x:
   Step size: stepi, stepj - 2's complement 7 bits (-64 to 63) Modulo size: modi, modj - unsigned 9 bits (1 to 512)
   Options controlled by st2:
   For each r_n register it should be defined if Modulo is used or not.
   For using modi or modj the relative m_n bit must be set.

   Assembler syntax: (r_n) , (r_n)+, (r_n)- , (r_n)+s
3. **ph** (the MSP of the **p** register) can be write only. The 32-bit **p register** is updated after a multiply operation and can be read only by transferring it to the ALU, that is, it can be moved into aX or be an operand for arithmetic and logic operations. When transferring it into the ALU, it is sign-extended to 36 bits. This enables the user to store and restore the **p** register.
4. The **p** register is used as a **source** operand, as one of the reg registers (e.g. in pacr instruction) or in multiply instructions, where the **p** register is added or subtracted from one of the accumulators. When using the **p** register as a source operand, it always means using the "**shifted p register**." Shifted **p** register means that the **p** register is sign- extended into 36 bit and then shifted as defined at the **sp** field, status register st1. In shift right the sign is extended, whereas in shift left a zero is appended into the LSB. The contents of the **p** register remain unchanged.
5. All move instructions using the accumulator (**aX**) as a destination are sign extended. All instructions which use the accumulator-low (**aXl**) as a destination, will clear the accumulator-high and the accumulator-extension. Therefore, they are sign extension suppressed.

   All instructions using the accumulator-high (**aXh**) as a destination, will clear the accumulator-low and are sign extended. An exception is mov direct address, aXh,{eu}, when moving data into accumulator-high can be controlled with sign extension or with sign extension suppressed (the accumulator-extension aXe is unaffected).
6. In all arithmetic operations between 16-bit registers and aX (36 bits), the 16-bit register will be regarded as the 16 low-order bits of a 36-bit operand with a sign extension in the Most-Order-Bits.
7. It is recommended that the flags be used immediately after the ALU operation or moved into ac operations. Otherwise, very careful programming is required (some flags may be changed in the meantime).
8. The condition field is an optional field; when the condition is missing then cond = true.
9. When transferring data into the hardware stack, the data is transferred to the **tos**, and the stack is pushed down one level. When transferring data out of the hardware stack, the data is copied to the destination, and the stack is popped one level.
10. ALU instruction is one of the following instructions: **add, sub, or, and, xor, cmp, addl, subl, addh, subh, moda, norm, mac, msu, sqra, sqrs.**

A.2 Instruction Set Summary

add

add to accumulator

Syntax: add operand, aX

Operand:
reg
#short immediate
##long immediate
(r_n)
direct address

Operation: aX + operand -> aX

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

addh

add to high accumulator

Syntax: addh operand, aX

Operand:
(r_n)
direct address
reg (except aX, p)

Operation: aX + operand*2^16 -> aX ; aXl is unaffected

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

addl

add to low accumulator

Syntax: addl operand, aX

Operand:
(r_n)
direct address
reg (except aX, p)

Operation: aX + operand -> aX (operand is sign-extension suppressed)

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

and

and accumulator

Syntax: and operand, aX

Operand:
reg
(r_n)
direct address
#short immediate
##long immediate

Operation:
```
If operand is aX or p
   aX(35:0) and operand -> aX(35:0)
If operand is short immediate
   aX(7:0) and operand -> aX(7:0)
   aX(15:8) -> aX(15:8)
   0 -> aX(35:16)
If operand is reg, (rn) or long immediate
   aX(15:0) and operand -> aX(15:0)
   0 -> aX(35:16)
```

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

bkrep

block repeat

Syntax: bkrep operand, add

Operand:
#short immediate
reg

Operation:
```
operand -> lc
1 -> lp status bit
```

Begins an interruptible block of instructions that is to be repeated operand + 1 (1..256) times.

Affects flags: No
Note: Address "add" must be temporary label

br

conditional branch

Syntax: br address [, cond]

Operation:
```
If condition then
   address -> pc
```

Affects flags: No

brr

relative conditional branch

Syntax: brr $offset address [, cond]

Operation:
```
If condition then
   pc + 1 + $offset address  ->  pc
```

Affects flags: No

call

conditional call subroutine

Syntax: call address [, cond]

Operation:
```
If condition then
   pc  -> tos
   address -> pc
```

Affects flags: No

calla

call subroutine at address specified by accumulator

Syntax: calla aXl

Operation:
```
pc -> tos
(aX) -> pc
```

Affects flags: No

callr

relative conditional call subroutine

Syntax: callr $offset address [, cond]

Operation:
```
If condition then
   pc  -> tos
   pc + 1 + $offset address ->  pc
```

Affects flags: No

clr

conditional clear accumulator

Syntax: `clr aX [, cond]`

Operation:
```
If condition then
   0 -> aX
```

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	-	-	*	-	-

clrr

conditional clear and round accumulator

Syntax: `clrr aX [, cond]`

Operation:
```
If condition then
   0x8000 -> aX
```

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	-	-	*	-	-

cmp

compare to accumulator

Syntax: `cmp operand, aX`

Operand:
```
reg
(rn)
direct address
#short immediate
##long immediate
```

Operation: `aX - operand`

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	*	*	*	*	-

copy

conditional copy accumulator

Syntax: `copy aX [, cond]`

Operation:
```
If condition then
   ay -> aX
```

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	-	-	*	-	-

dint

disable interrupts

Syntax: dint

Operation: 0 -> ie

Affects flags: No

divs

division step

Syntax: divs direct address, aX

Operation:
```
aX - (direct address)*2^15 -> ALU output
If ALU output < 0  then
   aX = aX * 2
else
   aX = ALU output * 2 + 1
```

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

eint

enable interrupts

Syntax: eint

Operation: 1 -> ie

Affects flags: No

lpg

load the page bits

Syntax: lpg #short immediate

Operation: #short immediate -> 8 low order bits of st1

Affects flags: No

mac

multiply and accumulate previous product

Syntax: mac operand1, operand2, aX

Operands:
y, direct address
y, (r_n)
y, reg (except aX, p)
(r_j), (r_i) (XRAM & YRAM)
(r_n), ##long immediate

Operation:
aX + shifted p -> aX
operand1 -> y
operand2 -> x
x * y -> p

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

moda

modify accumulator conditionally

Syntax: [moda] Function , aX [, cond]

Operation:
If condition then
aX is modified by 'Function'

Function:

shr	aX = aX >> 1
shl	aX = aX << 1
shr4	aX = aX >> 4
shl4	aX = aX << 4
ror	Rotate aX right through carry
rol	Rotate aX left through carry
not	aX = not(aX)
neg	aX = -aX
clr	aX = 0_
copy	aX = aX
rnd	aX = aX + 0x8000
pacr	aX = shifted p + 0x8000
clrr	aX = 0x8000

Affects flags: According to function, when condition is true.

modr

Modify r_n

Syntax: modr (r_n)

Operation: r_n is modified.

Affects flags:

Z	M	N	V	C	E	L	R
-	-	-	-	-	-	-	*

Note: R flag is set if r_n register is zero, otherwise cleared.

mov

move data

Syntax: mov soperand, doperand

Soperand, doperand:

reg , reg
reg , (r_n)
(r_n), reg
r_n , direct address
aXl , direct address
aXh , direct address
y , direct address
x , direct address
direct address , r_n
direct address , y
direct address , x
direct address , aX
direct address , aXl
direct address , aXh [, eu]
##long immediate , reg
#short immediate , aXl
#signed short immediate , aXh
#signed short immediate , r_n
#signed short immediate , y
#signed short immediate , x

Operation: soperand -> doperand

Affects flags: No effect when doperand is not ac, st0
No effect when soperand is not aXl, aXh
When soperand is aXl or aXh:

Z	M	N	V	C	E	L	R
-	-	-	-	-	-	*	-

When doperand is ac:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

If doperand is st0:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	*

movp

Move Program Memory

Syntax: movp soperand, doperand

Soperand, doperand:

(aXl), reg
(r_n) , (r_i)

Operation: soperand points to prom -> doperand

Affects flags: No effect when doperand is not ac, st0.

When doperand is ac:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

If the doperand is st0:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	*

mpy

multiply

Syntax: mpy operand1, operand2

Operands:

y , direct address	
y , (r_n)	
y , reg	(except aX, p)
(r_j) , (r_i)	(XRAM & YRAM)
(r_n) , ##long immediate	

Operation:
operand1 -> y
operand2 -> x
x * y -> p

Affects flags: No

mpys

multiply signed short immediate

Syntax: mpys y, #signed short immediate

Operation:
#signed short immediate -> x
x * y -> p

Affects flags: No

msu

multiply and subtract previous product

Syntax: msu operand1, operand2, aX

Operands:
y , direct address
y , (r_n)
y , reg (except aX, p)
(r_j) , (r_i) (XRAM & YRAM)
(r_n) , ##long immediate

Operation:
aX - shifted p -> aX
operand1 -> y
operand2 -> x
x * y -> p

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

neg

conditional negate accumulator

Syntax: neg aX [, cond]

Operation:
If condition then
-aX -> aX

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

nop

No Operation

Syntax: nop

Operation: No operation

Affects flags: No

norm

normalize accumulator

Syntax: norm aX, r_n

Operation:
```
If n = 0  (aX is not normalized) then
     aX = aX * 2
     rn is modified
else
    nop
    nop
```

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	*	*	*	*	*

not

conditional (bitwise logic) not accumulator

Syntax: not aX [, cond]

Operation:
```
If condition then
   not (aX) -> aX
```

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	-	-	*	-	-

or

or accumulator

Syntax: or operand, aX

operand:
```
reg
(rn)
direct address
#short immediate
##long immediate
```

Operation:
```
If operand is aX or p  then
   aX(35:0) or operand -> aX(35:0)
else
   aX(15:0) or operand -> aX(15:0)
   aX(35:16) -> aX(35:16)
```

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	-	-	*	-	-

pacr

round product

Syntax: pacr aX

Operation: shifted p + 0x8000 -> aX

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

rep

repeat next instruction

Syntax: rep operand

Operand: #short immediate
reg (except aX, p)

Operation: Begins a noninterruptible single word instruction loop, to be repeated operand + 1 (1..256) times.

Affects flags: No

ret

conditional return from subroutine

Syntax: ret [cond]

Operation:
```
If condition then
   tos -> pc
```

Affects flags: No

reti

return from interrupt

Syntax: reti

Operation:
```
tos -> pc
1 -> ie
```

Affects flags: No

rnd

conditional round accumulator

Syntax: rnd aX [, cond]

Operation: If condition then
aX + 0x8000 -> aX

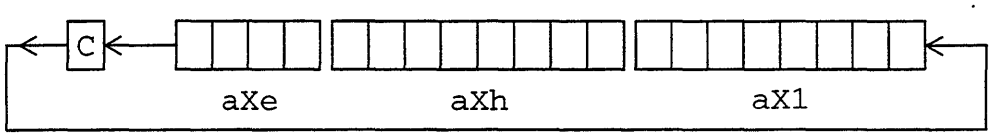
Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	*	*	*	*	-

rol

conditional rotate accumulator left

Syntax: rol aX [, cond]

Operation: If condition then



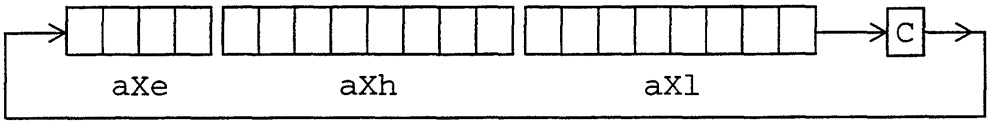
Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	-	*	*	-	-

ror

conditional rotate accumulator right

Syntax: ror aX [, cond]

Operation: If condition



Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	-	*	*	-	-

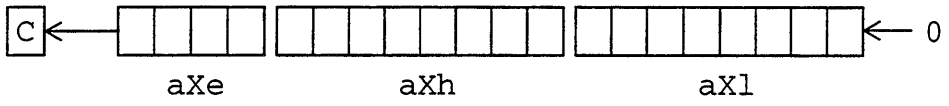
shl
sh14

conditional shift left accumulator 1 bit

conditional shift left accumulator 4 bits

Syntax:
```
shl aX
shl4 aX
```

Operation: If condition then



Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	*	*	*	-	-

Note: V is cleared if the operand being shifted could be represented in 35 bits for shl, in 31 bits for shl4, set otherwise.

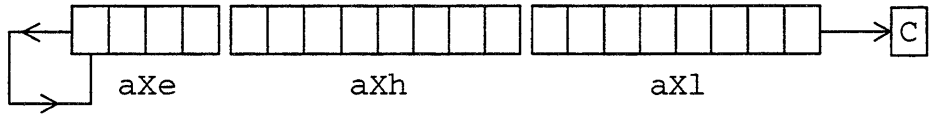
shr
shr4

conditional shift right accumulator 1 bit

conditional shift right accumulator 4 bits

Syntax:
```
shr aX
shr4 aX
```

Operation: If condition then



Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	0	*	*	-	-

sqr

square

Syntax: sqr operand

Operand: (r_n)
reg (except aX, p)
direct address

Operation: operand -> y
operand -> x
y * x -> p

Affects flags: No

sqra

square and accumulate previous product

Syntax: sqra operand, aX

Operand: (r_n)
reg (except aX, p)
direct address

Operation: aX + shifted p -> aX
operand -> y
operand -> x
y * x -> p

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

sqrs

square and subtract previous product

Syntax: sqrs operand, aX

Operand: (r_n)
reg (except aX, p)
direct address

Operation: aX - shifted p -> aX
operand -> y
operand -> x
y * x -> p

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

sub

subtract from accumulator

Syntax: sub operand, aX

Operand:
reg
(r_n)
direct address
#short immediate
##long immediate

Operation: aX - operand -> aX

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	*	*	*	*	-

subh

subtract from high accumulator

Syntax: subh operand, aX

Operand:
(r_n)
direct address
reg (except aX, p)

Operation: aX - operand*2^16 -> aX (aXl is unaffected)

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	*	*	*	*	-

subl

subtract from low accumulator

Syntax: subl operand, aX

Operand:
(r_n)
direct address
reg (except aX, p)

Operation: aX - operand -> aX (operand is not sign-extended)

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	*	*	*	*	-

trap

software interrupt

Syntax: trap

Operation:
```
pc -> tos
0x000A -> pc
Disable interrupts (int0 , int1).
```

Affects flags: No

xor

exclusive or accumulator

Syntax: xor operand, aX

operand:
```
reg
(rn)
direct address
#short immediate
##long immediate
```

Operation:
```
If operand is aX or p  then
   aX(35:0) xor operand -> aX(35:0)
else
   aX(15:0) xor operand -> aX(15:0)
   aX(35:16) -> aX(35:16)
```

Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	-	-	*	-	-

This page intentionally left blank.

APPENDIX B
Restrictions Checked By DM_ASM

B.1 Instruction Restrictions

The following list of restrictions on the use of DSP instructions are imposed by the DSP architecture, e.g. the pipe line mechanism or the interconnection of busses. These restrictions are checked by the assembler and their violation is reported. Self restrictions are restrictions imposed on the use of two operands of the same instruction, while forward restrictions are restrictions imposed on an instruction by subsequent instructions.

B.1.1 Self Restriction on ALU Instructions:

It is forbidden to use the same accumulator as both the source and destination in ALU register instructions.

B.1.2 Self Restriction on aX and p:

It is forbidden to use the aX and p registers as source in the following instructions:

`addh`, `addl`, `bkrep`, `mac`, `mpy`, `msu`, `rep`, `sqr`, `sqra`, `sqrs`, `subh` and `subl`

B.1.3 Self Restriction on Indirect mov Instructions:

Indirect moves where the source and destination operands are the same r_n register are forbidden.

B.1.4 Self Restriction on reg-to-reg mov Instructions:

Register to register moves where the source and destination operands are the same register are forbidden.

B.1.5 Self Restriction on ac-to-reg mov Instructions:

Register to register moves where aX is the source operand and non aX is the destination operand, are forbidden.

B.1.6 Self Restriction on p-to-reg mov Instructions:

Register to register moves, where p is the source operand and non aX is the destination operand, are forbidden.

B.1.7 Self Restriction on pc as Source Register:

It is forbidden to use the pc as source register in the following instruction: bkrep

B.1.8 Block restrictions (bkrep):

After the **bkrep** instruction:

1. The body of the loop can not contain bkrep and mov/p to lc.
2. The last and the next to last instruction in the loop can not be:
   `br, brr, call, calla, callr, mov/p to pc, rep, ret/i, trap.`
3. The next to last instruction may not use the lc register. If the body of the loop consists of only one instruction, it may not use the lc register.
4. It is forbidden to jump to the last address of the loop with:
   `br, brr, call, calla, callr, ret/i, mov/p to pc.`
   (Returns from interrupt routines with **reti** command are allowed at any time).

B.1.9 Forward Restriction on Moving Data to the pc:

After moving data to the pc (using the **mov** instruction), the next instruction must be **nop**.

B.1.10 Forward Restriction on Repeat Instructions:

After a **rep** instruction, the following single-word instructions may not be used:
`brr, calla, callr, movp, rep, ret, reti` and `trap.`

B.1.11 Forward Restriction on Repeat Instruction Types:

Two-word instructions may not be used after a **rep** instruction.

B.1.12 Forward Restriction on st0:

After ALU instructions or after an instruction where a0, a0l, a0h, a1, a1l, a1h, st0 is the destination operand, st0 can not be used as a source:
`add, addh, addl, and, cmp, or, mac, moda, msu, norm, sqra, sqrs, sub, subh, subl, xor.`

APPENDIX C
Internal Preprocessor Directives

.ERROR FreeText

The ERROR directive is the equivalent of the C "#error" directive. It displays the free text (the given argument) as a message to "stderr" and exits. This directive is used by the preprocessor itself and is not recommended for use by the programmer.

.LINE number ["filename"]

The LINE directive is used to control the line number and/or name of the current source file for the purpose of reporting errors. This directive is inserted by the preprocessor for the assembler and is not recommended for use by the programmer.

.X

The X directive is used to signify each line but the first of a multiple line macro expansion in order to keep the source line counter synchronized. This directive is inserted by the preprocessor for the assembler and is not recommended for use by the programmer.

This page intentionally left blank.

APPENDIX D
DM_ASM Error Messages

D.1 Macro Pre-Processor Error Messages:

"PineMPP Error L001: %s(%d): Unexpected number: %s"
"PineMPP Error L002: %s(%d): Unexpected %s\n"
"PineMPP Error L003: %s(%d): Unexpected <cr>\n"
"PineMPP Fatal error L004: %s(%d): Exceeded maximum nesting level of '%d'\n"
"PineMPP Fatal error L005: %s(%d): Can not open file '%s'\n"
"PineMPP Fatal error L006: %s(%d): Invalid .INCLUDE directive: %s\n"
"PineMPP Error L007: %s(%d): Invalid .LINE directive"
"PineMPP Error L008: %s(%d): Unexpected .ENDM directive\n"
"PineMPP Warning Y002: %s(%d): .MACRO redefinition of %s\n"
"PineMPP Fatal error Y003: %s(%d): Too many .MACROs: %s\n"
"PineMPP Fatal error Y004: %s(%d): Too long(%i) .MACRO definition: %s\n"
"PineMPP Warning Y005: %s(%d): .EQU redefinition of %s\n"
"PineMPP Fatal error Y006: %s(%d): Too many .MACROs: %s\n"
"PineMPP Fatal error Y007: %s(%d): Too long(%i) .MACRO definition: %s\n"
"PineMPP Warning Y008: %s(%d): .EQU redefinition of %s\n"
"PineMPP Fatal error Y009: %s(%d): Too many .EQUs: %s\n"
"PineMPP Fatal error Y010: %s(%d): Too long(%i) .EQU definition: %s\n"
"PineMPP Warning Y011: %s(%d): .EQU redefinition of %s\n"
"PineMPP Fatal error Y012: %s(%d): Too many .EQUs: %s\n"
"PineMPP Fatal error Y013: %s(%d): Too long(%i) .EQU directive: %s\n"
"PineMPP Error Y014: %s(%d): Can not purge symbol: %s\n"
"PineMPP Warning Y015: %s(%d): Undefined symbol: %s\n"
"PineMPP Warning Y025: %s(%d): Missing parameters\n"
"PineMPP Error Y026: %s(%d): %s\n"
"PineMPP Fatal error Y027: Illegal switch %c\n"
"PineMPP Internal error Y101: %s(%d): Too long input string: %s\n"
"PineMPP Internal error Y102: %s(%d)"
"PineMPP Internal error Y108: %s(%d): Unable to add symbol\n"

"PineMPP Internal error Y109: %s(%d): Unable to delete symbol\n"
"PineMPP Internal error Y110: %s(%d): Unable to find symbol\n"
"PineMPP Internal error Y111: %s(%d): Too many tokens in input string: %s\n"
"PineMPP Internal error Y112: %s(%d): Too long input string: %s\n"
"PineMPP Internal error Y113: %s(%d): Too many tokens in input string: %s\n"
"PineMPP Internal error Y114: %s(%d): Too long input string: %s\n"
"PineMPP Internal error Y115: %s(%d): Can not create symbol table\n"

D.2 Syntax Error Messages:

".TITLE %s\nTitle Directive not yet implemented\n"
"DW directive allowed only in DATA segment"
"External symbol definition '%s'" - Attempt to define a symbol previously declared as external.
"Illegal instruction"
"Illegal shift value" - SHR operator is out of the range 0 - 15.
"Illegal use of temporary symbol"
- Use temporary symbol in SHR operation.
"Invalid MODA function"
"Invalid instruction"
"Invalid operand for current directive"
"Invalid operand1, should be #UShort or Reg"
"Invalid operand1, should be #UShort"
"Invalid operand1, should be (Ax) or (Rn)"
"Invalid operand1, should be (Rn)"
"Invalid operand1, should be Address"
"Invalid operand1, should be Ax"
"Invalid operand1, should be AxL"
"Invalid operand1, should be Cond or nothing"
"Invalid operand1, should be Direct"
"Invalid operand1, should be Y"
"Invalid operand1, should be offset expression"
"Invalid operand1, should be one of (Rn), Y"
"Invalid operand1, should be one of Reg, (Rn), Direct"
"Invalid operand1, should be one of Reg, (Rn), Direct, ##Long"
"Invalid operand2, should be ##Long"
"Invalid operand2, should be #Short"
"Invalid operand2, should be (Ri) or ##Long"
"Invalid operand2, should be (Ri)"

"Invalid operand2, should be (Rn)"
"Invalid operand2, should be Address"
"Invalid operand2, should be Ax"
"Invalid operand2, should be Reg"
"Invalid operand2, should be condition"
"Invalid operand2, should be one of Reg, (Rn), Direct"
"Invalid operand2, should be one of RegPH, (Rn)"
"Invalid operand2, should be one of RegPH, (Rn), Direct"
"Invalid operand2, should be one of Rn*, Ax, AxL, AxH"
"Invalid operand3, should be Ax"
"Invalid operand3, should be EU"
"Invalid operand3, should be condition"
"Invalid relocation type"
- The relocation type of a symbolic expression is not one of the following:
  Dollar, absolute address, forward or backward reference for temporary symbol.

"Label redefinition '%s'"
"Missing ',' between operands"
"No operand for current instruction"
"One hashmark required"
"Segment name can not include '.'"
"Segment name larger than 8 characters"
"Segment name required"
"Symbol redefinition '%s'"
"Too many segments declared for directive"
- The identifier list of .EXTERN, .GLOBAL or .PUBLIC exceeds the max. of 10 identifiers.

"Two hashmarks required"
"Undefined Public symbol: %s"
"Undefined symbol: %s"
"Undefined temporary label '%s'"
"WARNING: Label \"%s\" truncated to \"%s\""
"invalid LINE directive"

D.3 Range Checking Errors:

"Number exceeds 16 bits"
"Number exceeds digit limit"
"Out of range. ##Long range is (-32768 to +32767)"
"Out of range. #Short range is (-128 to 127)"

"Out of range. #UShort range is (0 to 255)"
"Out of range. Address range is (0 to 65535)"
"Out of range. Direct range is (0 to 255)"
"Out of range. Offset range is (-63 to 64)"

D.4 Logical Error Messages:

"Cannot immediately resolve symbol"
- IMMEDOFFSET operation with symbol which is not yet defined.

"Relocation size conflict, symbol '%s'"
"Segment %s used for both Code and Data"
"Segment stack depleted"
- Attempt to use POPSEG directive when no segment name has been pushed to stack.

"Temporary label relocation size conflict, symbol '%s'"

D.5 File I/O Messages:

"Can not open temporary string file"
"Can not re-read r_temporary contents file"
"Can not re-read temporary contents2 file"
"Can not re-read temporary relocation file"
"Can not re-read temporary section contents"
"Can not re-read temporary string file"
"Can not re-write section contents"
"Can not re-write section relocation info"
"Can not re-write string table"
"Can not re-write temporary section contents"
"Can not update temporary relocation file"
"Can not write file header"
"Can not write section header no. %d"
"Can not write string table count"
"Can not write symbol table info"
"Can not write temporary string table info"
"FATAL ERROR: Unable to write to temporary relocation entry file\n"
"File Problem while resolving temporary symbols"
"Unable to write to temporary contents file"
"error: unable to open/create object file '%s'\n"

D.6 Memory Allocation Messages:

"Assembler could not allocate sufficient memory"
"FATAL ERROR: Out of heap space\n"
"Unable to allocate memory for label:- \"%s\""
"Unable to allocate memory for section header \"%s\""
"Unable to allocate memory for symbol:- \"%s\""
"Unable to allocate memory for temporary symbol:- \"%s\""
"Unable to create symbol table"

D.7 Limitations Messages:

"Segment '%s' exceeds module section count limit"
"More than %d temporary labels"
"Segment '%s' size greater than 64K"
"Segment stack size exceeded"
- More that 16 segment names has been pushed to stack by PUSHSEG directive.
"'%s' exceeds symbol limit"

D.8 Restrictions Messages:

"After MOV to PC next instruction must be NOP"
"Ax and P regs cannot be used in this instruction"
"Ax src/dst oprnd requires Ax dst/src oprnd"
"Bkrep end of loop address is not a label expression"
"Bkrep label expression invalid"
"Bkrep label not in same segment"
"Branch to end of BKREP loop"
"Cannot repeat BKREP, BRR, CALLR, MOVP, REP, and TRAP instructions"
"Cannot repeat two word instructions"
"Cannot use the same accumulator for both src and dst oprnd"
"Cannot use the same reg for both src and dst oprnd"
"Cannot use the same reg for both src and dst oprnd"
"End of segment encountered before checking forward restriction of previous instruction"
"End of segment encountered before terminating BKREP loop"
"Final instruction of BKREP extends beyond loop boundary"
"Illegal branch to end of BKREP loop from address 0x%4.4X"
"Illegal instruction at end of BKREP loop, i.e. branch instruction"
"Illegal instruction in body of BKREP loop, i.e. nested BKREP or MOV/P to LC"

"Illegal instruction preceding end of BKREP loop, i.e. branch instruction or MOV/P from LC"
"Invalid Bkrep End Of Loop Address"
"P src oprnd requires Ax dst oprnd"
"PC cannot be used as src oprnd"
"STO is invalid src oprnd"
"unknown"

D.9 Command Line Messages:

"Unrecognized switch '%s\n"

D.10 Internal Error Messages:

"Could not fetch symbol from symbol table"
"Could not find current data segment in symbol table"
"Could not retrieve symbol from symbol table"
"Error accessing symbol table"
"Internal error: StartBit > 16 on second word"
"Retrieving symbol hash table info"
"Symbol hash table full"
"Unable to enter/access file name in hash table"

D.11 Information/Report Messages:

"****Restriction Pass\n"
"Cannot close server while still connected to clients. Suggest that you close clients first."
"\nNo errors in Assembly.\n"
"\nTotal of %d assembly errors. No object file created.\n"
"\nTotal of %d assembly warnings.\n"

Adaptec, Inc.
691 South Milpitas Boulevard
Milpitas, CA 95035
Tel: (408) 945-8600
Fax: (408) 262-2533

P/N: 700174-011 Rev 3
Printed in U.S.A. RJ 12/95