Enhanced High-End Timer (NHET) Assembler User's Guide

User's Guide

Literature Number: SPNU490
October 2011

Contents

List of Figures

List of Tables

Read This First

About This Manual

The TI's Enhanced High-End Timer (NHET) module provides sophisticated timing functions for complex, real-time applications, such as automobile engine management or power-train management. These applications require measurement of information from multiple sensors and drive actuators with complex timing.

This manual describes the NHET assembler, tells how to use the assembler, and summarizes the NHET instruction set.

Notational Conventions

This document uses the following conventions.

The TI's Enhanced High-End Timer is abbreviated as the NHET.

Program listings, program examples, and interactive displays are shown in a special typeface. Examples use a **bold version** of the special typeface for emphasis; interactive displays use a **bold version** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

Here is a sample program listing:

```
.asg         1 , x
.loop
.byte        x*10h
.break       x == 4
.eval        x+1,  x
.endloop
```

In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of directive syntax:

.width *page width*
The directive is .width. This directive has one parameter, indicated by *page width*.

Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you do not enter the brackets themselves. Here is an example of a directive that has optional parameters:
.mexit *[parameter1 ... parametern]*

Braces ({ and }) indicate a list. The pipe symbol **|** (read as *or*) separates items within the list. Here is an example of a list:

{ * | *+ | *- }

This provides three choices: *, *+, or *-.

Unless the list is enclosed in square brackets, you must choose one item from the list.

Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

.byte *value*$_1$ *[, value2] ... [, value*$_n$*]*

This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Assembler Description

The NHET assembler translates assembly language source files into machine language object files for the NHET assembly source debugger. These files are in common object file format (COFF). The NHET assembler also produces *.hbj* files for use with the NHET simulator, as well as C header files. Source files contain the following assembly language elements.

Assembler directives are described in Chapter 2.

Assembly language instructions are described in Chapter 3.

Macro directives are described in Chapter 4.

This chapter gives an overview of the NHET assembler and how it fits into the development process for the assembly language tools, as well as information about how to use the NHET assembler.

1.1 Assembler Overview

The NHET assembler translates assembly language source files into machine language. Once the assembly source files have been translated, the NHET assembler can output a *.hnc* file to the host assembler, a *.h* and a *.c* header file to the host compiler, or a COFF *.hbj* file to the NHET simulator.

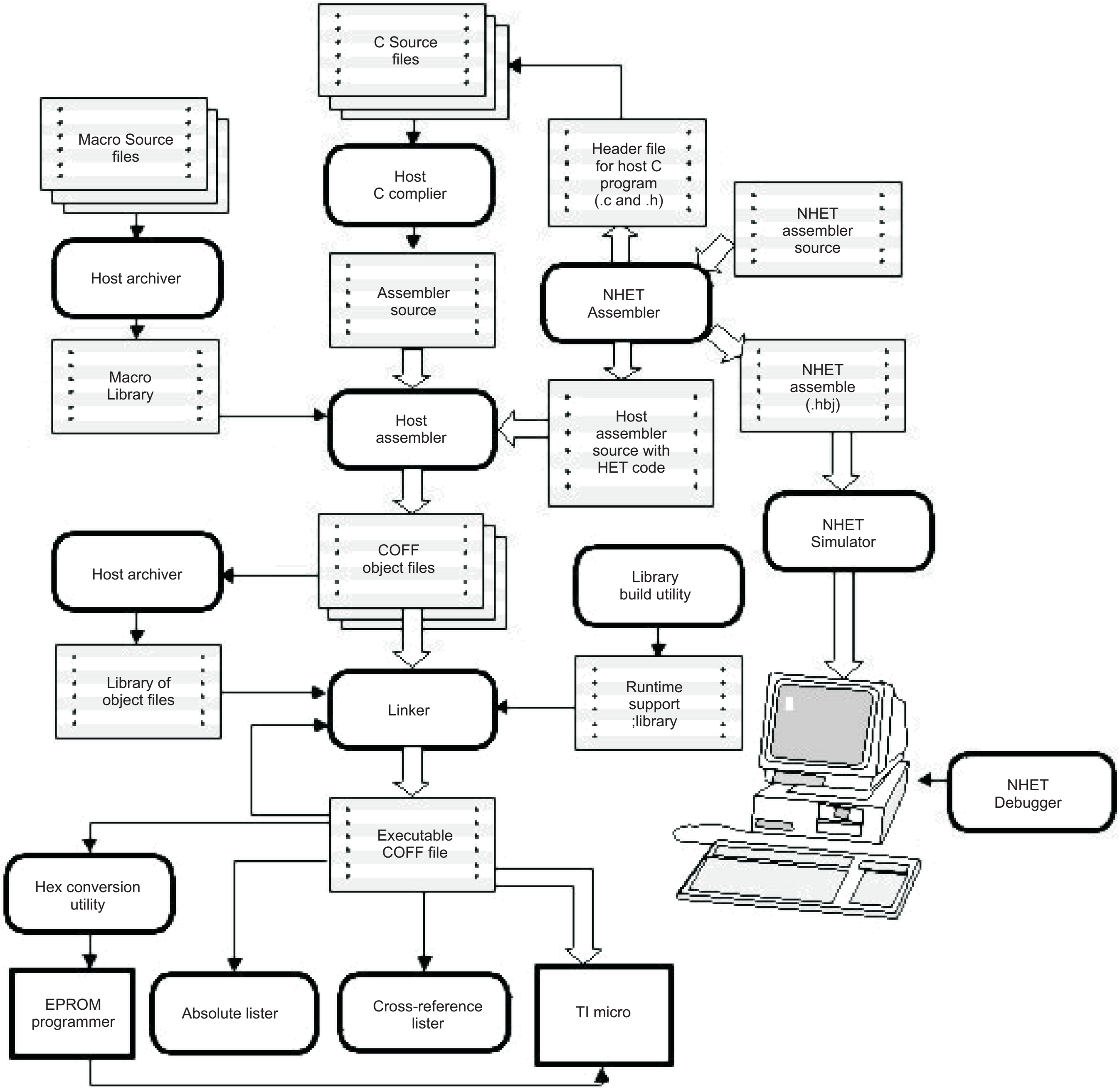
The two-pass NHET assembler does the following tasks:

- Processes the source statements in a text file to produce an object file
- Produces a source listing (if requested) and provides you with control over this listing
- Produces header files to support symbol, code, and structure type definitions, which can be used by C programs
- Produces output files that can be used by the host assembler to produce object files for the host processor
- Allows conditional assembly
- Supports macros, allowing you to define macros inline or in a library

1.2 The Assembler's Role in the Software Development Flow

Figure 1-1 illustrates how the NHET assembler works with the host assembly language tools in the software development flow. The NHET assembler accepts assembly language source files as input and outputs to either the host assembler or the NHET debugging tools or both. The NHET assembler can also output C header files that can be used by the host compiler. The gray area of the figure represents the main software development flow when using the NHET tools.

Figure 1-1. The HET Assembler in the Software Development Flow



The following list describes the tools shown in Figure 1-1.

NHET Assembler The **NHET assembler** translates NHET assembly language source files into ma chine language object files. The NHET assembler can generate the COFF object file (*.hbj*) for the NHET simulator, the *.hnc* file for the host assembler, and the *.h* and *.c* files for the host C compiler.

NHET Simulator The main purpose of the development process is to produce a module that can be executed in an **NHET target system**. You can use the NHET simulator to simulate the operation of the NHET target system and the NHET debugger to refine and correct your code.

NHET Debugger The **NHET debugger** is a programmer's interface that helps you to develop, test, and refine NHET assembly language programs. You can use the debugger as an interface for the software simulator.

Host Compiler The **host C compiler** accepts C source code and the C header file and produces assembly language source. See the appropriate C language tools user's guide for your device for an explanation of how to use the compiler.

Host Assembler The **host assembler** translates assembly language source files into machine language COFF object files. See the appropriate assembly language tools user's guide for your device for an explanation of how to use the assembler.

COFF allows you to divide your code into logical blocks, define your system's memory map, and then link code into specific memory areas. COFF also provides rich support for source-level debugging.

1.3 Invoking the Assembler

The general form of the NHET assembler invocation command is as follows:

hetp *[options] input file [output file]*

---

hetp is the command that invokes the assembler.

options identify the assembler options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command or the options can be given inside the input file (.het). Precede each option with a hyphen. You can combine single-letter options that do not have parameters: for example, -ls is equivalent to -l –s.

-c32	produces an output file containing assembler directives for the TI’s assembler.
-hc32	produces C header file (*.h*) and source file (*.c*) for the Texas Instruments TI’s C compiler (this option has to be used with -nx).
-i	specifies a directory where the assembler can find files named by the .copy, .include, and .mlib directives. The syntax for the -i option is -i*pathname*. You can specify up to 32 directories in this manner; each pathname must be preceded by the -i option. For more information on using the -i option, see Section 1.4.1.
-nx	specifies the “**x**”-th NHET module in the device. The valid value of x is 0-9. If given more than single digit, last digit is considered; e.g., 12 will be considered as 2. (this option has to be used with -hc32).
-l	(lowercase L) produces a listing file with the same name as the input file with a *.lst* extension.
-s	produces a COFF object file for the Texas Instruments NHET simulator (this option must be used in order to create a .hbj file for use with the NHET simulator).
-v2	NHET version 2 supports additional instructions (ADD, SUB, ADC, SBB, RCNT etc.) compared to NHET version 1. Please make sure which NHET version is implemented on the target device before using this option.

-x	produces a cross-reference table and appends it to the end of the listing file. If you do not request a listing file but use the -x option, the assembler creates a listing file automatically.
-AIDx.x	assembler ID, this option helps in verifying whether right version of NHET assembler is used. Assembler throws error if **x.x** in **AIDx.x** option does not match with Assembler version.

NOTE: If assembler "*options*" are given through input file (.het), the following points have to taken care:

1. Precede each option with a hyphen ("-") starting at first column of the line.
2. Each option has to be in separate line.
3. Do not use command line options and options through input file at the same time.

Example 1-1. NHET Assembler Input File With Options (Test.het)

```
;------------------------------------------
; Assembler option
;------------------------------------------
-hc32
-n0
-v2
-AID1.6
;------------------------------------------
start: mov32{
                 next=l02, reg=A, data=0f0fh, remote=01,
                 type=imtoreg, init=on
            }
l02:   mov32{
                 next=start, reg=B, data=00f1h,
                 remote=00, type=imtoreg, init=off
             }
```

Invoking Assembler > **hetp Test.het** (this is equivalent to **hetp –hc32 –n0 –v2 –AID1.6 Test.het** if options are not provided through input file – Test.het).

input file	name of the assembly language source file. If you do not supply an extension, the assembler uses the default extension *.asm*. If you do not supply an input filename, the assembler prompts you for one.
output file	names the output file that the assembler creates. The extension for the output file is dependent upon the options used when invoking the assembler. If no option is used, **.hnc** is the default file name extension.

-hnc	file name extension for output file containing NHET assembler directives
-lst	file name extension for output file that is a C program listing file
-h	file name extension for output file that is a C language header file
-c	file name extension for output file that is a C language source file
.hbj	file name extension for output file that is used by the NHET simulator

1.4 Naming Alternate Directories for Assembler Input

The .copy, .include, and .mlib directives tell the assembler to use code from other files. The .copy and .include directives tell the assembler to read source statements from another file, and the .mlib directive names a library that contains macro functions. Chapter 2 contains examples of the .copy, .include, and .mlib directives. The syntax for these directives is:

.copy ["]*filename*["]
.include ["]*filename*["]
.mlib ["]*filename*["]

The *filename* names a copy/include file that the assembler reads statements from or a macro library that contains macro definitions. The *filename* may be a complete pathname, a partial pathname, or a filename with no path information. The assembler searches for the file in:

1. The directory that contains the current source file. The current source file is the file being assembled when the .copy, .include, or .mlib directive is encountered.
2. Any directories named with the -i assembler option.
3. Any directories named with the A_DIR environment variable.

Because of this search hierarchy, you can augment the assembler's directory search algorithm by using the -i assembler option (described in Section 1.4.1) or the A_DIR environment variable (described in Section 1.4.2).

1.4.1 Using the -i Assmbler Option

The -i assembler option names an alternate directory that contains copy/include files or macro libraries. The format of the -i option is as follows:

hetp –i *pathname [other options] input filename*

You can use up to 32 -i options per invocation; each -i option names one pathname. In assembly source, you can use the .copy, .include, or .mlib directive without specifying path information. If the assembler does not find the file in the directory that contains the current source file, it searches the paths designated by the -i options.

For example, assume that a file called source.asm is in the current directory; source.asm contains the following directive statement:

```
.copy “copy.asm”
```

Assume the following paths for the copy.asm file:

MS-DOS| , Windows NT| , or Windows| 95 C:\470tools\files\copy.asm

SunOS| version 4.1x (or higher) or HP-UX| /470tools/files/copy.asm

Operating System	Enter
MS-DOS, Windows NT, or Windows 95	`hetp -ic:\470tools\files source.asm`
SunOS or HP-UX	`hetp -i/470tools/files source.asm`

If you invoke the assembler for your system as shown above, the assembler first searches for copy.asm in the current directory because source.asm (the input file) is in the current directory. Then the assembler searches in the directory named with the -i option.

1.4.2 Using the A_DIR Environment Variable

An environment variable is a system symbol that you define and assign a string to. The assembler uses the A_DIR environment variable to name alternate directories that contain copy/include files or macro libraries. The command syntax for assigning the environment variable is as follows:

Operating System	Enter
MS-DOS, Windows NT, or Windows 95	`set A_DIR= pathname1;pathname2`
SunOS or HP-UX	`setenv A_DIR "pathname1;pathname`

The *pathnames* are directories that contain copy/include files or macro libraries. You can separate the pathnames with a semicolon or with blanks. In assembly source, you can use the .copy, .include, or .mlib directive without specifying path information. If the assembler does not find the file in the directory that contains the current source file or in directories named by the -i option, it searches the paths named by this environment variable.

1.5 Source Statement Format

The NHET assembly language source programs consist of source statements that can contain assembler section directives, assembly language instructions, and comments. Source statement lines can be as long as the source file format allows, but the assembler reads up to 200 characters per line. If a line contains more than 200 characters, the assembler truncates the line and issues a warning. A single source statement can be spread over more than one line.

Following are examples of source statements:

```
Start: ECMP {
             reg = A,
             pin = CC1,
             action = SET,
             irq = ON,
             index = 3,
             angle_comp = OFF,
             data = 0FFFFh
             }
Step:  SCNT { next = label4, data = 65534, gapstart = 0AACEh, step
32}
```

A source statement can contain four ordered fields (label, mnemonic, operand list, and comment). The general syntax for source statements is as follows:

```
[label] mnemonic {
                 [operand list] [,] [;comment]
                 }
```

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks, tabs, or commas must separate each field.
- A mnemonic cannot begin in column 1 or it will be interpreted as a label.
- Comments are optional and can be interspersed within the instructions. Comments that begin in column 1 can begin with an asterisk (*) or a semi- colon (;), but comments that begin in any other column must begin with a semicolon. All characters following the semicolon or asterisk are ignored until the end of the line is reached.
- A source statement can be longer than one line
- A single line cannot be longer than 200 characters.
- Operands are enclosed within braces { }.

1.5.1 Label Field

Labels are optional for all assembly language instructions. When used, a label must begin in column 1 of a source statement. A label can contain up to 32 alphanumeric characters (A-Z, a-z, 0-9, _, and $). The first character of a label cannot be a number. A label can be followed by a colon (:); the colon is not treated as part of the label name. If you do not use a label, the first character in column 1 must be a blank, a semicolon, or an asterisk.

When you use a label, its value is the current value of the **section program counter** (SPC). The label points to the statement with which it is associated. For example, if you use the .byte directive to initialize several bytes, a label would point to the first byte. Example 1-2 shows the format for the labels Start, label1, Here, and There in assembler source statements.

Example 1-2. Label Format in Assembly Source Statements

```
----------------------------------------------------------
"label.asm"
----------------------------------------------------------
 .sect ".HETCODE", 04000h
 .HDA 020h
 Start .byte 0Ah, 03h, 07h, 0Dh
 label1 .equ $ ; $ provides the current value of the SPC
 Here:   .byte 3
 There:  .space 24
```

A label on a line by itself is a valid statement. The label assigns the current value of the SPC to the label; this is equivalent to the following directive statement:

```
label.equ    $ ;   $ Provides the current value of SPC
```

When a label appears on a line by itself, it points to the instruction on the next line. The SPC is not incremented.

```
Here:
.byte 3
```

1.5.2 Mnemonic Field

The mnemonic field follows the label field in a source statement. The mnemonic field cannot start in column 1; if it does, it will be interpreted as a label.

The mnemonic field can contain one of the following opcodes:

- Machine-instruction mnemonic (such as ECMP, SCMP, BCAP)
- Assembler directive (such as .copy, .list, .equ)
- Macro directive (such as .macro, .var, .mexit)

1.5.3 Operand Field

The operand field follows the mnemonic field and contains a list of operands. Typically an operand list has the following syntax:

```
ECMP {
       operand1     = { keywordb1 | value1 | label1 } [ , ] [;comment ]
       operand2     = { keywordb2 | value2 | label2 } [ , ] [;comment ]
       operand3     = { keywordb3 | value3 | label3 } [ , ] [;comment ]
       .
       .
       .
       operandn     = { keywordbn | valuen | labeln } [ , ] [;comment ]
       }
```

The list of operands is enclosed in bold braces **{ }**. The bold type indicates that you must type these braces as part of the syntax. Non-bold braces { } indicate a list of options from which you must choose one option. In the above example, you would choose between entering a keyword, a value, or a label. You do not type the nonbold braces.

In most cases, each operand corresponds to a single subfield within the 48-bit instruction format. If an operand in the instruction is optional, the default value for the corresponding subfield is zero. The operand *next* is the only operand that does not have a default value of zero. The default value for *next* is the section program counter (SPC) plus 1.

Optional operands and fields are indicated in this document by enclosing them in square brackets [ ].

Operands must be separated by spaces, commas, or a new line.

1.5.4 Comment Field

A comment can begin in any column and extends to the end of the source line. A comment can contain any ASCII character, including blanks. Comments are printed in the assembly source listing (there is a limit of 200 characters per line), but they do not affect the assembly.

A source statement that contains only a comment is valid. If the comment begins in column 1, it can start with a semicolon (;) or an asterisk (*). Comments that begin anywhere else on the line must begin with a semicolon. The asterisk identifies a comment only if it appears in column 1.

1.6 Output File Formats

The NHET program is loaded into the NHET device memory by the host CPU during initialization. The output of the NHET assembler consists of source code for the host processor so that the host can perform this process. Table 1-1 lists the options that produce source code for the assemblers and compilers supported by the NHET assembler.

Table 1-1. Options That Produce Source Code for NHET-Supported Tools

If you want to create this type of file	Use this option r
Coff object file for the NHET simulator (.hbj)	**-s**
C header file (.h) and C source file (.c) for theTI's Compiler	**-hc32**

Example 1-3 is a NHET source code program. Example 1-4 and Example 1-5 (a) and (b) are output files that are generated by the NHET assembler from the NHET source code fragment in Example 1-3. Example 1-6 is an example of an NHET listing file output.

Example 1-3. NHET Source Code Program (Test.het)

```
start: mov32
            {
                 next=l02,
                 reg=A,
                 data=0f0fh,
                 remote=01,
                 type=imtoreg,
                 init=on
            }
l02: mov32
            {
                 next=start,
                 reg=B,
                 data=00f1h,
                 remote=00,
                 type=imtoreg,
                 init=off
            }
```

Example 1-4. NHET Assembler Output Object File (.hnc) Using the -c32 Option

```
.sect ".HETCODE"
            .word 0x00002401h
            .word 0x00000040h
            .word 0x0001E1E0h
            .word 0x00000000h
            .word 0x00000400h
            .word 0x00000002h
            .word 0x00001E20h
            .word 0x00000000h
```

Example 1-5. NHET Assembler Output C Source and C Header File (.c and .h) Using the -hc32 Option

```
a) .c output file :test.c
#include "std_het.h"
#include "test.h"
#include "define.h"

HET_MEMORY const HET_INIT1_PST[2] =
{
         /* start_1 */
         {
               0x00002401,
               0x00000040,
               0x0001E1E0,
               0x00000000
         },
         /* l02_1 */
         {
               0x00000400,
               0x00000002,
               0x00001E20,
               0x00000000
         }
};

(b) .h output file
#define HET_start_1 (e_HETPROGRAM1_UN. Program1_ST. start_1)
#define pHET_start_1 0
#define HET_l02_1 (e_HETPROGRAM1_UN. Program1_ST.l02_1)
#define pHET_l02_1 1
HET_MEMORY       Memory1_PST[2];
typedef union
{
         struct
         {
              MOV32_INSTRUCTION start_1;
              MOV32_INSTRUCTION l02_1;
         } Program1_ST;
} HETPROGRAM1_UN;
extern HETPROGRAM1_UN e_HETPROGRAM1_UN;
```

Example 1-6. NHET Assembler Output Listing File Using the -l Option

```
NHET Assembler 4.1 Wed May 6 14:24:43 1998 Copyright (c)
2009,2010 Texas Instruments Incorporated

test.het                                          PAGE 1

HA 2000
       1 0020 0000 0040 0001 E1E0            start: mov32
       0020 0000 0000 0000 2401
       2                                 {
       3                                 next=l02,
       4                                 reg=A,
       5                                 data=0f0fh,
       6                                 remote=01,
       7                                 type=imtoreg,
       8                                 init=on
       9                                 }
       HA 2010
       10 0020 0000 0002 0000 1E20          l02: mov32
       0020 0000 0000 0000 0400
       11                                {
       12                                next=start,
       13                                reg=B,
       14                                data=00f1h,
       15                                remote=00,
       16                                type=imtoreg,
       17                                init=off
       18                                }

No Errors, No Warnings
```

Assembler Directives

Assembler directives supply data to the program and control the assembly process. Assembler directives enable you to do the following:

- Reserve space in memory
- Control the appearance of listings
- Initialize memory
- Assemble conditional blocks
- Specify libraries from which the assembler can obtain macros
- Examine symbolic debugging information

This chapter lists the directives and describes them according to function.

2.1 Directives Summary

The NHET assembler supports two NHET-specific directives as well as a number of generic directives. Table 2-1 summarizes the generic assembler directives supported by the NHET assembler. The NHET-specific directives are:

.HETCODE
.HDA

The syntax for the **.HETCODE** section directive is as follows:

sect " **.HETCODE**" **,** *address*

The assembler directive .HETCODE is used to associate the NHET code and data into the memory location corresponding to the address you specify. This has no effect on the actual address in which the host linker will place the code; however, this directive does change the listing file. NHET code and data in the listing file appear to be assigned to the address specified by the .HETCODE directive.

The assembler directive **.HDA** is used to associate the NHET code to specific NHET device addresses. The assembler pads any unused instruction words with 0s to fill in gaps between instructions.

The syntax for the .HDA directive is as follows:

.HDA address

Table 2-1. Generic Assembler Directives Summary

Mnemonic and Syntax	Description
(a) Directives that initialize constants Section 2.2	
.byte *value$_1$ [,value$_2$] ... [,value$_n$]*	Initialize one or more successive bytes in the current section
.space *size in bytes*	Reserve *size* bytes in the current section; a label points to the beginning of the reserved space
.bes *size in bytes*	Reserve *size* bytes in the current section; a label points to the end of the reserved space
(b) Directives that reference other files Section 2.4	
.copy ["]*filename*["]	Include source statements from another file
.include ["]*filename*["]	Include source statements from another file

Table 2-1. Generic Assembler Directives Summary (continued)

Mnemonic and Syntax	Description
.mlib ["]*filename*["]	*Define* macro library
(c) Directives that control conditional assembly Section 2.5	
.break [*well-defined expression*]	End .loop assembly if *well-defined expression* is true. When using the .loop construct, the .break construct is optional
.else	Assemble code block if the .if *well-defined expression* is false. When using the .if construct, the .else construct is optional
.elseif *well-defined expression*	Assemble code block if the .if *well-defined expression* is false and the .elseif condition is true. When using the .if construct, the elseif construct is optional
.endif	End .if code block
.endloop	End .loop code block
.if *well-defined expression*	Assemble code block if the *well-defined expression* is true
.loop [*well-defined expression*]	Begin repeatable assembly of a code block; the loop count is determined by the *well-defined expression*
(d) Macro directives Section 4.3	
.mexit [*parameter1...parametern*]	Go to .endm
.endm [*parameter1...parametern*]	End macro definition
(e) Directives that send user-defined messages to the output device Section 2.7	
.emsg *string*	Send user-defined error messages to the output device
.mmsg *string*	Send user-defined messages to the output device
.wmsg *string*	Send user-defined warning messages to the output device
(f) Directives that define symbols at assembly time Section 2.6	
symbol **.equ** *value*	Equate *value* with *symbol*
.eval *well-defined expression, substitution symbol*	Perform arithmetic on numeric *substitution symbol*
symbol **.set** *value*	Equate *value* with *symbol*
.var sym_1 [,sym_2] ... [,sym_n]	Define up to 32 local macro substitution symbols
(g) Directives that format the output listing Section 2.3	
.drlist	Enable listing of all directive lines (default)
.drnolist	Suppress listing of certain directive lines
.fclist	Allow false conditional code block listing (default)
.fcnolist	Suppress false conditional code block listing
.length *page length*	Set the page length of the source listing
.list	Restart the source listing
.mlist	Allow macro listings and loop blocks (default)
.mnolist	Suppress macro listings and loop blocks
.nolist	Stop the source listing
.option $option_1$ [,$option_2$] ... [,$option_n$]	Select output listing options; available options are A, B, F, M, T, W, and X
.page	Eject a page in the source listing
.sslist	Allow expanded substitution symbol listing
.ssnolist	Suppress expanded substitution symbol listing (default)
.tab *size*	Set tab spacing in listing output
.title "*string*"'	Print a title in the listing page heading
.title "*string*"'	Print a title in the listing page heading
.width *page width*	Set the page width of the source listing to *page width*

2.2 Directives That Initialize Constants

Several directives assemble values for the current section:

The **.byte** directive places one or more 8-bit values into consecutive bytes of the current section.

The **.space** directive reserves a specified number of bytes in the current section. The assembler fills these reserved bytes with 0s. When you use a label with .space, it points to the *first* byte of the reserved block.

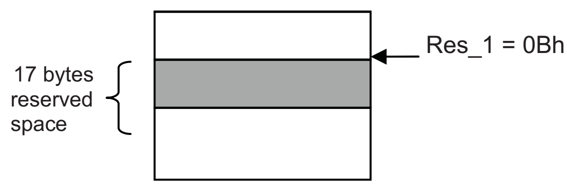
Figure 2-1 shows the .space directive. Assume the following code has been assembled for this example:

```
450007      0100            .word 100h,200h
0009        0200
6000B             Res_1:    .space 17
47001C      000F            .word 15
```

Res_1 points to the first byte of the 17 bytes in the space reserved by .space.

The **.bes** directive reserves a specified number of bytes in the current section. The assembler fills these reserved bytes with 0s. When you use a label with .bes, it points to the *last* byte of the reserved block.

Figure 2-1. The .space Directive



Example 2-1. Using Directives That Initialize Constants

```
          .title ""NHET Assembler Validation - .byte,.space,.bes"

fpt1 .space 4
fpt2 .space 1
x    .byte  1
lpt  .bes   4
y    .set   0

    MOV32   {brk=on, next=fpt1, remote=lpt, type=imtoreg, control=OFF, z_cond=on,
            init = on,reg = A,data=0,hr_data=0}
    MOV32   {brk=on, next=fpt2, remote=lpt, type=imtoreg, control=OFF, z_cond=on,
            init = on, reg = A, data=0,hr_data=0}
```

2.3 Directives That Format the Output Listing

Macros, substitution symbols, and conditional assembly directives may hide information. You may need to see this hidden information, so the macro language supports an expanded listing capability.

By default, the assembler shows macro expansions and false conditional blocks in the list output file. You may want to turn this listing off or on within your listing file. Seven sets of directives enable you to control the listing of this information.

Macro and loop expansion listing

.mlsit Macro and loop expansion listing

.mnolist suppresses the listing of macro expansions and .loop/ .endloop blocks.

For macro and loop expansion listing, .mlist is the default.

False conditional block listing

.fclist causes the assembler to include in the listing file all conditional blocks that do not generate code (false conditional blocks). Conditional blocks appear in the listing exactly as they appear in the source code.

.fcnolist suppresses the listing of false conditional blocks. Only the code in conditional blocks that actually assemble appears in the listing. The .if, .elseif, .else, and .endif directives do not appear in the listing.

For false conditional block listing, .fclist is the default.

Substitution symbol expansion listing

.sslist expands substitution symbols in the listing. This is useful for debugging the expansion of substitution symbols. The expanded line appears below the actual source line.

.ssnolist turns off substitution symbol expansion in the listing.

For substitution symbol expansion listing, .ssnolist is the default.

Directive listing

.drlist causes the assembler to print to the listing file all directive lines.

.drnolist suppresses the printing of the following directives in the listing file:

.break	.fclist	.mlist	.sslist	.width
.emsg	.fcnolist	.mmsg	.ssnolist	.wmsg
.eval	.length	.mnolist	.var	

You can use the .drlist directive to turn the listing on again.

For directive listing, .drlist is the default.

Page format

.length controls the page length of the listing file. You can use this directive to adjust listings for various output devices.

.tab defines tab size.

.title supplies a title that the assembler prints at the top of each page.

.width controls the page width of the listing file. You can use this directive to adjust listings for various output devices.

Output listing

.nolist prevents the assembler from printing selected source statements in the listing file.

.list turns the listing on again.

.page causes a page eject in the output listing.

Listing file

.option controls certain features in the listing file. This directive has the following operands:

A	turns on listing of all directives and data, and subsequent expansions, macros, and blocks.
B	limits the listing of .byte directives to one line.
F	resets the B and M directives (turns off the limits of and M).
M	turns off macro expansions in the listing.
X	produces a cross-reference listing of symbols.

Example 2-2. Using Directives That Output Format Listing - False Condition

```
           .width 200
           .title "NHET Assembler Validation - .fclist(by defautl enabled),.fcnolist"
           .fcnolist
           .include enable.asm

Mov_Mac .macro LOOP1, LR_CNT_ADR, Data_Val
MOV32      {brk=on, next=1, remote=LR_CNT_ADR, type=imtoreg, control=OFF,
             z_cond=on,init = on,reg = A, data =Data_Val, hr_data=1}

RCNT       {brk= OFF, next = 1, control = OFF, divisor = 0xff, data = 0x1ffff}

PCNT       {hr_lr=high, brk=on, next=1,reqnum=1,control= on, request=NOREQ,
             pin = 1, prv = on, type = FALL2RISE, period =0x1FFFF, irq=OFF,
             data = 1}

ADC        {src1= ZERO, src2 = ZERO, dest = NONE, rdest = NONE, brk= OFF,
             next=2, remote = 0, control = OFF, init = OFF, smode = LSL, scount =
             1, data = 0x1FFFF}

         .endm
                      .if enable1
                         Mov_Mac 0,1,2
                      .elseif enable2
                         Mov_Mac 3,4,5
                      .else
                         Mov_Mac 6,7,8
                      .endif
```

Example 2-3. Using Directives That Output Format Listing - Substitution Symbol

```
           .width 200
           .title " NHET Assembler Validation - .sslist,.ssnolist (by default
```

Example 2-3. Using Directives That Output Format Listing - Substitution Symbol (continued)

```
                     enabled)"
           .sslist
           .include enable.asm

Mov_Mac .macro LOOP1, LR_CNT_ADR, Data_Val
MOV32      {brk=on, next=1, remote=LR_CNT_ADR, type=imtoreg, control=OFF,
             z_cond=on,init = on,reg = A, data =Data_Val, hr_data=1}

RCNT       {brk= OFF, next = 1, control = OFF, divisor = 0xff, data = 0x1ffff}

PCNT       {hr_lr=high, brk=on, next=1,reqnum=1,control= on, request=NOREQ,
             pin = 1, prv = on, type = FALL2RISE, period =0x1FFFF, irq=OFF,
             data = 1}

ADC        {src1= ZERO, src2 = ZERO, dest = NONE, rdest = NONE, brk= OFF,
             next=2, remote = 0, control = OFF, init = OFF, smode = LSL, scount =
             1, data = 0x1FFFF}

         .endm
                      .if enable1
                         Mov_Mac 0,1,2
                      .elseif enable2
                         Mov_Mac 3,4,5
                      .else
                         Mov_Mac 6,7,8
                      .endif
```

Example 2-4. Using Directives Listing

```
           .width 200
           .title "NHET Assembler Validation - .drlist(by defautl
                   enabled),.drnolist"
           .drnolist
           .sslist
           .fcnolist
           .mnolist
           .include enable.asm

Mov_Mac .macro LOOP1, LR_CNT_ADR, Data_Val
MOV32      {brk=on, next=1, remote=LR_CNT_ADR, type=imtoreg, control=OFF,
             z_cond=on,init = on,reg = A, data =Data_Val, hr_data=1}

RCNT       {brk= OFF, next = 1, control = OFF, divisor = 0xff, data = 0x1ffff}

PCNT       {hr_lr=high, brk=on, next=1,reqnum=1,control= on, request=NOREQ,
             pin = 1, prv = on, type = FALL2RISE, period =0x1FFFF, irq=OFF,
             data = 1}

ADC        {src1= ZERO, src2 = ZERO, dest = NONE, rdest = NONE, brk= OFF,
             next=2, remote = 0, control = OFF, init = OFF, smode = LSL, scount =
             1, data = 0x1FFFF}

         .endm
                      .if enable1
                         Mov_Mac 0,1,2
                      .elseif enable2
                         Mov_Mac 3,4,5
                      .else
                         Mov_Mac 6,7,8
                      .endif
```

Example 2-5. Using Directives Output Format Listing - Page Format

```
          .title "NHET Assembler Validation - .length,.width,.tab,.title"
          .width 200
          .drlist
          .tab 10
          .length 200

MOV32     {brk=on, next=1, remote=2 , type=imtoreg, control=OFF, z_cond=on,
            init = on, reg = A, data =0, hr_data=1}
```

Example 2-6. Using Directives Output Format Listing - Output listing

```
          .width 200
          .length 200
          .title "NHET Assembler Validation- .page,.list,.nolist"
          .page
          .nolist

MOV32     {brk=on, next=1, remote=2, type=imtoreg, control=OFF, z_cond=on, init
            = on, reg = A, data =0, hr_data=1}
```

Example 2-7. Directive That Ref Other Files

```
          .width 200
          .length 200
          .title "NHET Assembler Validation- .page,.list,.nolist"
          .page
          .nolist

MOV32     {brk=on, next=1, remote=2, type=imtoreg, control=OFF, z_cond=on, init
            = on, reg = A, data =0, hr_data=1}
```

Example 2-8. Directive Output Format Listing - Macro

```
          .title "NHET Assembler Validation - .mlist,.mnolist"
          .list
          .mnolist

NADDR         .equ 10
RADDR         .set 20
 MOV32     {brk=on, next=NADDR, remote=RADDR, type= imtoreg, control=OFF,
           z_cond=on, init = on, reg = A, data =0, hr_data=0}
             .eval NADDR+10,NADDR
 RCNT      {brk= OFF, next = NADDR, control = OFF, divisor = 0xff, data = 0x1ffff}
             .eval NADDR+10,NADDR
 PCNT      {hr_lr=high, brk=on, next=NADDR, reqnum=1, control= on,
           request=NOREQ, pin = 1, prv = on, type = FALL2RISE, period =0x1FFFF,
           irq=OFF, data = 1}

 MACRO_LS    .macro arg
             .var Np,Nr
             .eval arg+10,Np
             .eval Np+10,Nr

 MOV32    {brk=on, next=Np, remote=Nr, type=IMTOREG, control=OFF, z_cond=on,
          Init = on, reg = A, data =0, hr_data=0}
             .endm

         MACRO_LS 0
         MACRO_LS 1
```

2.4 Directives That Reference Other Files

These directives supply information for or about other files that may be used in the assembly of the current file:

The **.copy** and **.include** directives tell the assembler to begin reading source statements from another file. When the assembler finishes reading the source statements in the copy/include file, it resumes reading source statements from the current file. The statements read from a copied file are printed in the listing file; the statements read from an included file are not printed in the listing file.

The **.mlib** directive supplies the assembler with the name of an archive library that contains macro definitions. When the assembler encounters a macro that is not defined in the current module, it searches for it in the macro library specified with .mlib.

Example 2-9. Using Directives That Reference Other Files

```
        .title "NHET Assembler Validation - .copy,.include,.mlib"
        .copy enable.asm
        .include global.asm

MOV32       {brk=on, next=enable1, remote=enable2, type=imtoreg, control=OFF,
            z_cond=on, init = on, reg = A, data=0, hr_data=0}
MOV32       {brk=on, next=value1, remote=value2, type=imtoreg, control=OFF,
            z_cond=on, init = on, reg = A, data=0, hr_data=0}
```

2.5 Directives That Enable Conditional Assembly

Conditional assembly directives enable you to instruct the assembler to assemble certain sections of code according to a true or false evaluation of an expression. Two sets of directives allow you to assemble conditional blocks of code:

The **.if/.elseif/.else/.endif** directives tell the assembler to conditionally assemble a block of code according to the evaluation of an expression.

.if *well-defined if expression*	marks the beginning of a conditional block and assembles code if the *well-defined if expression* is false
.elseif *well-defined else/if*	marks a block of code to be assembled the *well-defined if expression* is false
.else	marks a block of code to be assembled the *well-defined if expression* is false
.endif	marks the end of a conditional block and terminates the block.

The **.loop/.break/.endloop** directives tell the assembler to repeatedly assemble a block of code according to the evaluation of an expression.

.loop *well-defined loop expression*	marks the beginning a repeatable block of code. The optional expression evaluates to the loop count.
.break *well-defined break expression*	tells the assembler to continue to repeatedly assemble when the *well-defined break expression* is false and to go to the code immediately after end loop when the expression is true.
.endloop	marks the end of a repeatable block.

Example 2-10. Using Directives That Control Conditional Assembly - if

```
       .title "NHET Assembler Validation - .if,.elseif,.else,.endif"
       .copy enable.asm
Mov_Mac .macro LOOP1,LR_CNT_ADR,Data_Val

 MOV32      {brk=on, next=1, remote=LR_CNT_ADR, type=imtoreg, control=OFF,
            z_cond=on, init = on, reg = A, data =Data_Val, hr_data=1}

 RCNT       {brk= OFF, next = 1, control = OFF, divisor = 0xff, data = 0x1ffff}

 PCNT       {hr_lr=high, brk=on, next=1, reqnum=1, control= on, request=NOREQ, pin =
            1, prv = on, type = FALL2RISE,period =0x1FFFF,irq=OFF,data = 1}

 ADC        {src1= ZERO, src2 = ZERO, dest = NONE, rdest = NONE, brk= OFF,
            next=2, remote = 0, control = OFF, init = OFF, smode = LSL, scount = 1,
            data = 0x1FFFF}
            .endm
                        .if enable1
                          Mov_Mac 0,1,2
                        .elseif enable2
                          Mov_Mac 3,4,5
                        .else
                          Mov_Mac 6,7,8
                        .endif
```

Example 2-11. Using Directives That Control Conditional Assembly - loop

```
         .title "NHET Assembler Validation :- .loop,.break,.endloop"
x .set 0
y .set 0
w .set 0

Mov_Mac  .macro LOOP1,LR_CNT_ADR,Data_Val
MOV32    {brk=on, next=1, remote=LR_CNT_ADR, type= imtoreg, control=OFF,
         z_cond=on, init = on, reg = A, data =Data_Val, hr_data=1}
RCNT     {brk= OFF, next = 1, control = OFF, divisor = 0xff, data = 0x1ffff}
PCNT     {hr_lr=high, brk=on, next=1, reqnum=1, control= on, request=NOREQ, pin =
         1, prv = on, type = FALL2RISE,period =0x1FFFF,irq=OFF,data = 1}
ADC      {src1= ZERO, src2 = ZERO, dest = NONE, rdest = NONE, brk= OFF,
         next=2, remote = 0, control = OFF, init = OFF, smode = LSL, scount = 1,
         data = 0x1FFFF}
         .endm
                    .loop
                      .eval x+1,y
                            .eval x+2,w
                            Mov_Mac x,y,w
                    .break x == 2
                    .eval x+1,x
                    .endloop
```

2.6 Directives That Define Symbols at Assembly Time

Assembly-time symbol directives equate meaningful symbol names to constant values or strings.

The **.eval** directive evaluates a well-defined expression, translates the results into a character string, and assigns the character string to a substitution symbol. This directive is most useful for manipulating counters:

```
.loop
.byte         x*10h
.eval         x+1, x
.endloop
```

The **.set** and **.equ** directives set a constant value to a symbol. The symbol is stored in the symbol table and cannot be redefined. In the following example, bval is set to 0100h:

```
.bval  .set   0100h
.bytebval,   , bval*2, bval+12 b
              Bval
```

The **.set** and **.equ** directives produce no object code. The two directives are identical and can be used interchangeably.

The **.var** directive defines up to 32 local macro substitution symbols per macro. The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed into the program as parameters, and they are lost after expansion. The .var directive is used in Example 2-12.

Example 2-12. Using Subscripted Substitution Symbols to Redefine an Instruction

```
Cmpx            .macro  x
                .var tmp
                .asg :x(1):, tmp
                .if $symcmp(tmp, "E") == 0
                .asg :x(2,$symlen(x)):, tmp
                ECMP { reg = T, data = tmp, index = 7 }
                .elseif $symcmp(tmp, "S") == 0
                .asg :x(2,$symlen(x)):, tmp
                SCMP {pin = tmp, index = 7, data = 0, action = SET }
                .elseif $symcmp(tmp, "M") == 0
                .asg :x(2,$symlen(x)):, tmp
                MCMP { reg = tmp, index = 6, data = 0, order = DATA_GE_REG
                }
                .else
                .emsg "Bad Macro Parameter"
                .endif
                .endm

                cmpx E100          ;macro call
                cmpx SIF2          ;macro call
                cmpx MA            ;macro call
```

Example 2-13. Using Subscripted Substitution Symbols to Redefine an Instruction

```
          .title "NHET Assembler Validation - .set, .equ, .eval,.var(should be
used with a macro only)"
          .list

 NADDR          .equ 10
 RADDR          .set 20

  MOV32         {brk=on, next=NADDR, remote=RADDR, type= imtoreg,
                control=OFF, z_cond=on, init = on, reg = A, data =0, hr_data=0}
          .eval NADDR+10,NADDR
  RCNT          {brk= OFF, next = NADDR, control = OFF, divisor = 0xff, data =
                0x1ffff}
          .eval NADDR+10,NADDR
  PCNT          {hr_lr=high, brk=on, next=NADDR ,reqnum=1, control= on,
                request=NOREQ, pin = 1, prv = on, type = FALL2RISE,period
                =0x1FFFF, irq=OFF,data = 1}

 MACRO_LS .macro arg
          .var Np,Nr
          .eval arg+10,Np
          .eval Np+10,Nr
  MOV32        {brk=on, next=Np, remote=Nr, type=IMTOREG, control=OFF,
               z_cond=on, init = on, reg = A, data =0, hr_data=0}
          .endm

        MACRO_LS 0
```

2.7 Directives That Send User-Defined Messages to the Output Device

These three directives enable you to define your own error and warning messages:

- The **.emsg** directive sends error messages to the standard output device. The .emsg directive generates errors in the same manner as the assembler, incrementing the error count and preventing the assembler from producing an object file.
- The **.mmsg** directive sends assembly-time messages to the standard output device. The .mmsg directive functions in the same manner as the .emsg and .wmsg directives but does not set the error count or the warning count. It does not affect the creation of the object file.
- The **.wmsg** directive sends warning messages to the standard output device. The .wmsg directive functions in the same manner as the .emsg directive but increments the warning count rather than the error count. It does not affect the creation of the object file.

Example 2-14. Using Directives That Send User Defined Message

```
          .title "NHET Assembler Validation - .emsg, .wmsg, .mmsg"
          .copy enable.asm

          .if enable1

MOV32         {brk=on, next=1, remote=LR_CNT_ADR, type=imtoreg,
                control=OFF, z_cond=on, init = on, reg = A, data =Data_Val,
                hr_data=1}
          .mmsg "Enable1 is a non zero value"
          .elseif enable2

RCNT          {brk= OFF, next = 1, control = OFF, divisor = 0xff, data = 0x1ffff}

          .wmsg "Enable1 is a zero value and Enable2 is non zero value"
          .else

PCNT          {hr_lr=high, brk=on, next=1,reqnum=1, control= on,
                request=NOREQ, pin = 1, prv = on, type = FALL2RISE, period
                =0x1FFFF, irq=OFF, data = 1}
          .emsg "Enable1 and Enable2 has a zero value"

          .endif
```

Instruction Set

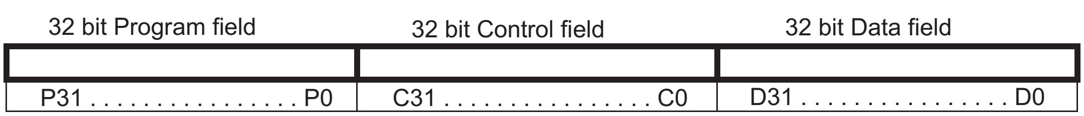
This chapter summarizes the NHET instruction set. Included are descriptions of the instruction format, the instruction fields and subfields, an explanation of the abbreviations used throughout the instruction set summary, flags and interrupt capabilities for the NHET assembler, and detailed information about each instruction in the instruction set. The instructions are presented alphabetically.

NOTE: Assembler supports Pseudo Instruction **DJNZ**. When **DJNZ** instruction is used the assembler generates DJZ instruction opcodes with Next address and Conditional address swapped. This feature is supported from assembler version 1.6. In versions before 1.6 same opcode are generated for DJNZ and DJZ instruction.

3.1 Instruction Format

The instructions for the NHET assembler are 96 bits wide. This wide format allows you to fetch the instruction opcode and data in one system cycle. Each instruction is organized in three 32-bit fields the program field, the control field, and the data field, as shown in Figure 3-1.

Figure 3-1. Instruction Fields



These 3 fields allow you to obtain and manipulate timing information, event counts, and angle values. During program execution, the control and data fields in RAM can be modified by the timer or by the CPU. The program field is never modified; when you have finished developing your code, this field can be converted into ROM.

The first eight bits of the program field specify the next instruction to be executed. These eight bits allow the instruction content to monitor the program flow instead of a program counter monitoring the program flow. The subfields for the program field, the control field, and the data field are listed in Table 3-1, Table 3-2, and Table 3-3, respectively.

Table 3-1. Program Field Subfields

Subfield	Width	Description
Next program address	9	defines the address next instruction in the program flow
Opcode	4	defines the operation code for the current instruction
Remote address	4	defines the four most significant bits for the remote address pointer for MOV32, MOV64, ADM32, DADM64, and ADCNST instruction
Count mode	2	selects the count instruction sharing the same opcode
Angle count	1	selects count on new angle in the CNT instruction
Register select	2	selects the A, B or T register for the arithmetic logic unit (ALU)
Index enable	1	specifies the index enable bit in the BCAP instruction
Capture enable	1	specifies the capture enable in CNT instruction
Save subtraction	1	saves the results of the subtraction of the MCMP instruction to register
Angle compare	1	specifies the angle compare bits for the ECMP and MCMP instructions
Interrupt enable	1	enables a CPU interrupt from the current instruction

Table 3-1. Program Field Subfields (continued)

Subfield	Width	Description
If enable	1	enables the use of internal flag for CNT BR and WCAP instructions
Pin select	2	selects the input pin for PCNT instruction
Period/pulse select	2	selects the period or the pulse duration measure for the PCNT instruction
Edge select	1	specifies the edge to detected in the ACNT instruction
Sted width	2	defines the step value in the SCNT instruction
SI/SO	1	defines direction (in or out) of the SHFT instruction

Table 3-2. Control Field Subfields

Subfield	Width	Description
Remote address	9	defines the four least significant bits of the remote address pointer for the MOV32, MOV64, ADM32, DADM64, ADCNST instruction
Previous	1	stores the previous state of the selected pin the CNT, WCAP and BR instruction
Count condition	3	specifies the counter increment conditions for the CNT instructions
Branch condition	3	specifies the branch condition in BR instruction
Capture condition	3	specifies the capture condition in the WCAP instruction
Compare mode	2	selects the compare instructions using the same opcode
Restart enable	1	specifies the restart enable for the SCMP instruction
SL/SR	1	selects the direction (left or right) for the SHFT instruction
Shift control	3	selects the shift condition for the SHFT instruction
Opposite action	1	defines the opposite pin action for the PWCNT, MOV32 and compare instruction
Index	4	defines the program address jump
Pin select	5	defines the pin used in the related instruction
Register select	2	selects the A,B or T register of the ALU
Pin action	1	defines the Pin action for PWCNT and MOV64 instruction
Interrupt enable	1	enables a CPU interrupt request from the current instruction
Reset flag	1	enables reset of the acceleration flag, deceleration flag and gap flag in the MOV32 and ADM32 instruction
Move type	3	defines the source and destination for MOV32 and ADM32 instructions
Maximum count	25	defines the maximum counter value for the CNT instruction
Period/pulse count	25	defines the value in the PCNT instruction
Gap start	25	defines the start valued of a gap in the SCNT instruction
Gap end	25	defines the end value of a gap in the ACNT instruction
Unconditional branch	1	forces an unconditional branch regardless of the select branch condition

Table 3-3. Data Field Subfields

Subfield	Width	Description
Compare value	25	stores a 16 bit comparison value for the ECMP, MCMP and SCMP instructions
Counter value	25	stores a 16 bit counter value for all counter instructions
Request number	3	defines the number of the request line (0,1,….,7) to trigger either the TU or the DMA
Request	2	Allows to select between no request and quiet request

3.2 Notational Conventions for the Instruction Descriptions

The instruction set presents each instruction separately. Each instruction description begins with the syntax, followed by a graphical representation of the format that shows the three instruction fields and each field's subfields. Following the graphical representation are preset bit values, descriptions of the operands shown in the syntax, a text description of how the instruction works, and a code example that uses the instruction. The appropriate instruction used in the code example is written in bold.

Table 3-4 alphabetically lists the symbols used throughout the rest of this chapter and describes the meaning of each symbol.

Table 3-4. Notations and Symbols Used in the Instruction Set Summary

Symbol	Definition
{ }	Curly braces indicate an entry that includes a list of items from which you must choose one item. Pipe symbols (|) are located between the choices within the curly braces.
[ ]	Square brackets identify an optional parameter. If you use an optional parameter, you specify the Information typed between the brackets; you do not enter the brackets themselves.
|	A pipe symbol indicates that you can choose between the parameters on either side of the symbol.
A	Register A in register file
B	Register B in register file
Bold	Bold text indicates an entry that must be typed in exactly as shown.
CC	Capture/compare pin
Italics	Italic text indicates the type of parameter to be entered. For example, *label* indicates that a label, such as my_code or start_here, is to be entered. The words in italics themselves are not entered.
IC	Input capture pin
IF	Internal flag
OC	Output capture pin
SCI	Serial communication interface
SPI	Serial peripheral interface
T	Register T in the register file

3.3 Alphabetical Summary of Instructions

Table 3-5 lists all of the instructions for the NHET assembler; the remainder of this chapter describes each of these instructions alphabetically.

Table 3-5. NHET Assembler Instructions

Abbreviation	Instruction name	Opcode	Sub-Opcode	Cycles
ACMP	Angle compare	Ch	-	1
ACNT	Angle count	9h	-	2
ADCNST	Add constant	5h	-	2
ADD	Add	4h	C[25:23]=001, C5 = 1	1-3
ADC	Add with carry	4h	C[25:23]=011, C5 = 1	1-3
ADM32	Add Move 32	4h	C[25:23]=000, C5 = 1	1 or 2
AND	Bitwise And	4h	C[25:23]=010, C5 = 1	1-3
APCNT	Angle Period Count	Eh	-	1 or 2
BR	Branch	Dh	-	1
CNT	Count	6h	-	1 or 2
DADM64	Data Add Move 64	2h	-	2
DJZ	Decrement and Jump if Zero	Ah	P7-6] = 10	1
ECMP	Equality compare	0h	C[6-5] = 00	1
ECNT	Event count	Ah	P[7-6] = 01	1
MCMP	Magnitude compare	0h	C[6] = 1	1

Table 3-5. NHET Assembler Instructions (continued)

Abbreviation	Instruction name	Opcode	Sub-Opcode	Cycles
MOV32	Move 32	4h	C[25:23]=000, C[5] = 0	1 or 2
MOV64	Move 64	1h	-	1
OR	Bitwise Or	4h	C[25:23]=100, C5 = 1	1-3
PCNT	Pulse/Period count	7h	-	1
PWCNT	Pulse width Count	Ah	P[7-6]=11	1
RADM64	Register Add Move 64	3h	-	1
RCNT	Ratio Count	Ah	P[7-6]=00, P[0]=1	3
SCMP	Sequence Compare	0h	C[6-5] = 01	1
SCNT	Step count	Ah	P[7-6] = 00, P[0] = 0	3
SHFT	Shift	Fh	C3=0	1
SUB	Subtract	4h	C[25:23]=101, C5 = 1	1-3
SBB	Subtract with carry	4h	C[25:23]=110, C5 = 1	1-3
WCAP	Software capture word	Bh	-	1
WCAPE	Software capture word and Event Count	8h	-	1
XOR	Bitwise XOR and Shift	4h	C[25:23] , C5 = 1	1-3

3.4 Flags and Interrupt Capable Instructions

Table 3-6 lists all the flags for the NHET assembler. Table 3-7 shows which instructions are capable of generating SW interrupts.

Table 3-6. NHET Assembler Flags

Abbreviation	Flag Name	Set/Reset by	Used by
C	Carry Flag	ADD, ADC, AND, OR, SUB, SBB, XOR, RCNT	BR
N	Negative Flag	ADD, ADC, AND, OR, SUB, SBB, XOR	BR
V	Overflow Flag	ADD, ADC, AND, OR, SUB, SBB, XOR	BR
Z	Z flag	ADD, ADC, SUB, SBB, AND, OR, XOR, SCNT, SHFT, CNT, APCNT, PCNT, ACNT, RCNT	ACMP, ECMP, SCMP, ACNT, BR, SHFT, MCMP, MOV32, RCNT
X	X flag	ACMP	SCMP
SWF 0-1	Step width flag	SCNT	ACNT
NAF	New Angle Flag	ACNT	NAF global
NAF global	New Angle flag (global)	HWAG or NAF	CNT, ECNT, BR, ACMP, ECMP
ACF	Acceleration flag	ACNT	SCNT, ACNT
DCF	Deceleration flag	ACNT	SCNT, ACNT
GPF	Gap flag	ACNT	APNT, ACNT

Table 3-7. Interrupt Capable Instructions

Interrupt capable instruction		Non-interrupt capable instruction	
ACMP	ECMP	ADCNST	OR
SCMP	MCMP	ADM32	SUB
CNT	ECNT	DADM32	SBB
ACNT	APCNT	MOV32	XOR
PWCNT	PCNT	MOV64	RCNT
DJZ	WCAP	RADM64	ADD
WCAPE	SHFT	SCNT	ADC

Table 3-7. Interrupt Capable Instructions (continued)

Interrupt capable instruction	Non-interrupt capable instruction
BR	AND

3.5 Abbreviations, Encoding Formats and Bits

Abbreviations marked with a star (*) are available only on specific instructions.

U Reading a bit marked with U will return an indeterminate value.

BRK Defines the software breakpoint for the device software debugger.

Default: OFF

Location: Program field [22]

NEXT Defines the program address of the next instruction in the pro- gram flow. This value may be a label or an 8-bit unsigned integer.

Default: Current instruction plus 1

Location: Program field [21:13]

reqnum* Defines the number of the request line (0,1,..,7) to trigger either the HTU or the DMA.

Default: 0

Location: Program field [25:23]

request* Allows to select between no request (NOREQ), request (GENREQ) and quiet request (QUIET).

Default: No request

Location: Control Field [28:27]

Table 3-8. Request Bit Field Encoding Format

Request	C[28]C[27]		To HTU	To DMA
NOREQ	0	0	no request	no request
	1	0		
GENREQ	0	1	request	request
QUIET	1	1	quiet request	no request

REMOTE Determines the 8-bit address of the remote address for the instruction.

Default: Current instruction plus 1

Location: Program field [8:0]

CONTROL Determines whether the immediate data field is cleared when it is read. When the bit is not set, reads do not clear the immediate data field.

Default: OFF

Location: Control field [26]

En_pin_action Determines whether the selected pin is ON so that the action occurs on the chosen pin.

Default: OFF

Location: Control field [22]

Cond_addr

Conditional address (optional): Defines the address of the next instruction when the condition occurs.

Default: Current address plus 1

Location: Control field [21:13]

PIN

Pin Select: Selects the pin on which the action occurs. Enter the pin number.

Default: Pin 0

Location: Control field [12:8] except PCNT

The format CC{pin number} is also supported.

Table 3-9. PIN Encoding Format

MSB				LSB	Description
0	0	0	0	0	Select HET 0
0	0	0	0	1	Select HET 1
(Each pin may be selected by writing its number in binary.)					
1	1	1	1	0	Select HET 30
1	1	1	1	1	Select HET 31

REG*

Register select: Selects the register for data comparison and storage.

Default: No register (None)

Location: Control field [2:1] except CNT

Table 3-10. Register Bit Field Encoding Format

Register	Reg Ext C[7]	C[2]	C[1]
A	0	0	0
B	0	0	1
T	0	1	0
None	0	1	1
R	1	0	0
S	1	0	1
Reserved (None)	1	1	0
Reserved (None)	1	1	1

The register bits field could be placed either in the Program field (CNT) or in the control field (all others' instructions use register field).

†The Ext Reg field applies only to: ACMP, ADD, ADC, ADM32, AND, DADM64, ECMP, ECNT, MCMP, MOV32, MOV64, OR, RADM64, SHFT, SUB, SBB, WCAP and WCAPE instructions.

ACTION

(2 Action Option) Either sets or clears the pin.

Default: Clear

Location: Control Field [4]

Table 3-11. Register Bit Field Encoding Format

Action	C[4]
Clear	0
Set	1

Action* (4 Action Option) Either sets, clears, pulse high or pulse low on the pin. Pulse high occurs when the pin is set on the compare and toggles at the overflow.

Default: Clear

Location: Control Field [4:3]

Table 3-12. PIN Action Bit Field (4 options)

Action	C[4]	C[3]
Clear	0	0
Set	0	1
Pulse Low	1	0
Pulse High	1	1

Bit C[4] is also called enable pin action and C[3] is also called opposite pin action.

hr_lr* Specifies high/low data resolution. If the hr_lr field is high, the instructionimplements the hr_data field (when the action is carried out on a high resolution pin). If the hr_lr field is low, the hr_data field is ignored.

Default: HIGH

Location: Program Field [8]

Table 3-13. High-Low Resolution Bit Field

hr_lr	Prog. field [8]
Low	0
High	1

Prv* Specifies the initial value defining the previous pin-level bit for the first edge detect performed by the instruction. The edge detect is performed by comparing the current pin value to the value stored in the previous pin-level bit. A value of ON sets the previous pin-level bit to 1. A value of OFF sets the initial value of the previous (prv) bit to 0. After the initial comparison, the value of the prv bit is set or reset by the system.

Default: OFF

Location: Control Field [25]

cntl_val* Available for DADM64, MOV64, and RADM64, this instruction allows the user to specify the replacement value for the remote control field.

comp_mode* Specifies the compare mode. This field is used with the 64-bit move instructions. This field ensures that the sub-opcodes are moved correctly.

Default: ECMP

Location: Control Field [6:5]

Table 3-14. Comp_mode Bit Field

Action	C[6]	C[5]	order
ECMP	0	0	
SCMP	0	1	
MCMP1	1	0	REG_GE_DATA
MCMP2	1	1	DATA_GE_REG

Table 3-15. Sub-Opcode Encoding for Arithmetic / Bitwise Logical Instructions

Instruction	Description	Sub Opcode Value [(1)] C[25:23], C[5]	Operation
ADD	Add	0 0 1 1	rdest = dest = src1 + src2
ADC	Add with Carry	0 1 1 1	rdest = dest = src1 + src2 + C
ADC	Add with Carry	0 1 1 1	rdest = dest = src1 + src2 + C
OR	Bitwise Logical Or	1 0 0 1	rdest = dest = src1 | src2
SUB	Subtract	1 0 1 1	rdest = dest = src1 - src2
SBB	Subtract with Borrow	1 1 0 1	rdest = dest = src1 - src2 - C
XOR	Bitwise Logical Exclusive-Or	1 1 1 1	rdest = dest = src1 ^ src2

(1) Opcode 4 is also shared with ADM32 (sub op. 0001) and MOV32 (sub op. 0000)

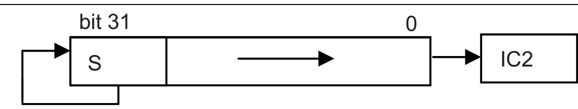
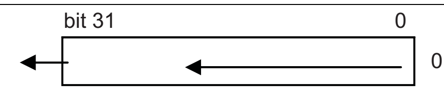
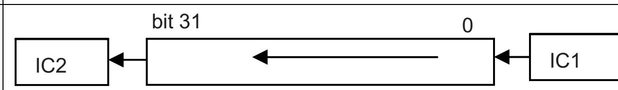
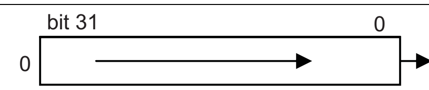
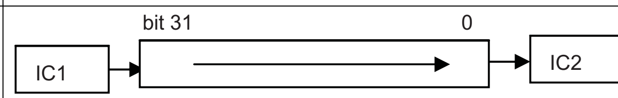
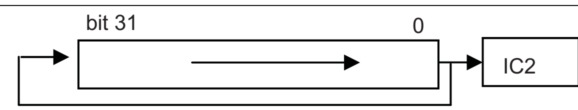
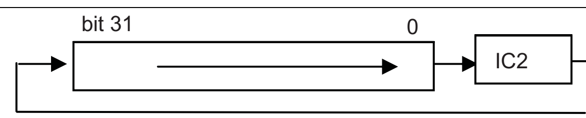
Table 3-16. Source 1 and Source 2 Register Encoding

Source Selected	Source 1 C[22:19]	Source 2 C[18:16]
Constant 32-bit all Zeros	0 0 0 0	0 0 0
Immediate Data Field	0 0 0 1	0 0 1
Register A	0 0 1 0	0 1 0
Register B	0 0 1 1	0 1 1
Register R	0 1 0 0	1 0 0
Register S	0 1 0 1	1 0 1
Register T	0 1 1 0	1 1 0
Constant 32-bit all Ones	0 1 1 1	1 1 1
Remote Data Field	1 0 0 0	n/a
Remote Program Field P[8:0]	1 0 0 1	n/a
	1 0 1 0	n/a
	1 0 1 1	n/a
Reserved (behaves as Constant 32-bit Zero)	1 1 0 0	n/a
	1 1 0 1	n/a
	1 1 1 0	n/a
	1 1 1 1	n/a

Table 3-17. Destination Encoding

Reg. Imm Destination	C[7], C[2:1]	Remote Destination	C[4:3]
Register A	0 0 0	None	0 0
Register B	0 0 1		
Register T	0 1 0	Remote Data Field D[31:0]	0 1
None	0 1 1		
Register R	1 0 0	Remote Program Field P[8:0]	1 0
Register S	1 0 1		
Immediate Data Field	1 1 0	Reserved - Behaves as None	1 1
Reserved (behaves as none)	1 1 1		

Table 3-18. Shift Encoding

Shift Type	C[15:14] smode	Operation Illustrated
No Shift Applied	0 0 0	n/a - no shift
ASR-Arithmetic Shift Right	0 0 1	
LSL-Logical Shift Left	0 1 0	
CSL-Carry Shift Left	0 1 1	
LSR-Logical Shift Right	1 0 0	
CSR-Carry Shift Right	1 0 1	
RR - Rotate Right	1 1 0	
CRR – Carry Rotate Right	1 1 1	

Macro Language

The assembler supports a macro language that enables you to create your own instructions. This is especially useful when a program executes a particular task several times. The macro language lets you:

- Define your own macros and redefine existing macros
- Simplify long or complicated assembly code
- Access macro libraries created with the host archiver
- Define conditional and repeatable blocks within a macro
- Manipulate strings within a macro
- Control expansion listing

4.1 Using Macros

Programs often contain routines that are executed several times. Instead of repeating the source statements for a routine, you can define the routine as a macro, then call the macro in the places where you would normally repeat the routine. This simplifies and shortens your source program.

If you want to call a macro several times but with different data each time, you can assign parameters within a macro. This enables you to pass different information to the macro each time you call it. The macro language supports a special symbol called a substitution symbol, which is used for macro parameters. See Section 4.5 for more information.

Using a macro is a three-step process:

Step 1: Define the macro. You must define macros before you can use them in your program. There are two methods for defining macros:

Macros can be defined at the beginning of a *source file* or in a .include/.copy file. See Section 4.3 for more information.

Macros can also be defined in a *macro library*. A macro library is a collection of files in archive format created by the host archiver. Each member of the archive file (macro library) may contain one macro definition corresponding to the member name. You can access a macro library by using the .mlib directive. For more information, see Section 4.4.

Step 2: Call the macro. After you have defined a macro, call it by using the macro name as a mnemonic in the source program. This is referred to as a *macro call*.

Step 3: Expand the macro. The assembler expands your macros when the source program calls them. During expansion, the assembler passes arguments by variable to the macro parameters, replaces the macro call statement with the macro definition, and then assembles the source code. By default, the macro expansions are printed in the listing file. You can turn off expansion listing by using the .mnolist directive. For more information, see Section 2.3.

When the assembler encounters a macro definition, it places the macro name in the opcode table. This redefines any previously defined macro, library entry, directive, or instruction mnemonic that has the same name as the macro. This allows you to expand the functions of directives and instructions, as well as to add new instructions.

4.2 Macro Directives Summary

The following tables summarize the macro directives.

Table 4-1. Creating Macros

Mnemonic and Syntax	Description	Section
macname **.macro** [*parameter*$_1$] [, ... , *parameter*$_n$]	Include source statements from another file	Section 4.3
.mlib ["]*filename*["]	Identify library containing macro definitions	Section 4.4
.mexit [*parameter1...parametern*]	Go to .endm	Section 4.3
.endm [*parameter1...parametern*]	End macro definition	Section 4.3

Table 4-2. Manipulating Substitution Symbols

Mnemonic and Syntax	Description	Section
.asg ["]*character string*["], *substitution symbol*	Assign character string to substitution symbol	Section 4.5.1
.eval *well-defined expression, substitution symbol*	Perform arithmetic on numeric substitution symbols	Section 4.5.1
.var *sym*$_1$ [,*sym*$_2$] ... [,*sym*$_n$]	Define local macro symbols	Section 4.5.6

Table 4-3. Conditional Assembly

Mnemonic and Syntax	Description	Section
.if *well-defined expression*	Begin conditional assembly	Section 4.6
.elseif *well-defined expression*	Optional conditional assembly block	Section 4.6
.else	Optional conditional assembly block	Section 4.6
.endif	End conditional assembly	Section 4.6
.loop [*well-defined expression*]	Begin repeatable block assembly	Section 4.6
.break [*well-defined expression*]	Optional repeatable block assembly	Section 4.6
.endloop	End repeatable block assembly	Section 4.6

Table 4-4. Producing Assembly-Time Messages

Mnemonic and Syntax	Description	Section
.emsg	Send error message to standard output	Section 4.8
.wmsg	Send warning message to standard output	Section 4.8
.mmsg	Send assembly-time message to standard output	Section 4.8

Table 4-5. Formatting the Listing

Mnemonic and Syntax	Description	Section
.fclist	Allow false conditional code block listing (default)	Section 2.3
.fcnolist	Suppress false conditional code block listing	Section 2.3
.mlist	Allow macro listings (default)	Section 2.3
.mnolist	Suppress macro listings	Section 2.3
.sslist	Allow expanded substitution symbol listing	Section 2.3
.ssnolist	Suppress expanded substitution symbol listing (default)	Section 2.3

4.3 Defining Macros

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file or in a .include/.copy file; they can also be defined in a macro library. For more information, see Section 4.4.

Macro definitions can be nested, and they can call other macros, but all elements of the macro must be defined in the same file. Nested macros are discussed in Section 4.9.

A macro definition is a series of source statements in the following format:

macname **.macro** [*parameter*$_1$] [, ... , *parameter*$_n$]
model statements or macro directives
[**.mexit**]
.endm

Macname	names the macro. You must place the name in the source statement's label field. Only the first 128 characters of a macro name are significant. The assembler places the macro name in the internal opcode table, replacing any instruction or previous macro definition with the same name.
.macro	is a directive that identifies the source statement as the first line of a macro definition. You must place .macro in the opcode field.
[*parameters*]	are optional substitution symbols that appear as operands for the .macro directive. Parameters are discussed in Section 4.5.
model statements	are instructions or assembler directives that are executed each time the macro is called.
macro directives	are used to control macro expansion.
[**.mexit**]	is a directive that functions as a goto .endm". The .mexit directive is useful when error testing confirms that macro expansion will fail and completing the rest of the macro is unnecessary.
.endm	terminates the macro definition.

Example 4-1 shows the definition, call and expansion of a macro.

Example 4-1. Macro Definition, Call, and Expansion

Macro definition: The following code defines a macro, ADCNST3, with three parameters:

```
1 ADCNST3.macro arg1, arg2, arg3
2 ADCNST { data = arg1 dest = arg2 min_off = arg3 }
3 .endm
4
```

Macro call: The following code calls the ADCNST3 macro with three arguments:

```
5.ADCNST3 0FFFFh, 21h, 0AAAh
```

Macro expansion: The following code shows the substitution of the macro definition for the macro call. The assembler passes the arguments (supplied in the macro call) by variable to the parameters (substitution symbols).

```
          HA 2000
1         0152 1AAA FFFF FFFF ADCNST { data = 0FFFFH dest = 21H min_off
= 0AAAH }
```

If you want to include comments with your macro definition but do not want those comments to appear in the macro expansion, use an exclamation point to precede your comments. If you do want your comments to appear in the macro expansion, use an asterisk or semicolon. See Section 4.8 for more information about macro comments.

Example 4-2. Using Macro Directives

```
          .title "NHET Assembler Validation - .macro,.mexit,.endm"

 Mov_Mac .macro LOOP1,LR_CNT_ADR,Data_Val

  MOV32   {brk=on, next=LOOP1, remote=LR_CNT_ADR, type=imtoreg,
          control=OFF, z_cond=on, init = on, reg = A, data =Data_Val,
```

Example 4-2. Using Macro Directives (continued)

```
        hr_data=1}

RCNT    {brk= OFF, next = 1, control = OFF, divisor = 0xff, data = 0x1ffff}
        .mexit
        .mmsg "Coming out of Macro"
PCNT    {hr_lr=high, brk=on, next=1, reqnum=1, control= on,
        request=NOREQ, pin = 1, prv = on, type = FALL2RISE, period
        =0x1FFFF, irq=OFF, data = 1}
ADC     {src1= ZERO, src2 = ZERO, dest = NONE, rdest = NONE, brk=
        OFF, next=2, remote = 0, control = OFF, init = OFF, smode = LSL,
        scount = 1, data = 0x1FFFF}
        .endm

      Mov_Mac 0,1,2
```

4.4 Macro Libraries

One way to define macros is by creating a macro library. A macro library is a collection of files that contain macro definitions. You must use the host archiver to collect these files, or members, into a single file (called an archive). Each member of a macro library contains one macro definition. The files in a macro library must be unassembled source files. The macro name and the member name must be the same, and the macro filename's extension must be .asm. For example:

Macro Name	Filename in Macro Library
Simple	simple.asm
add3	add3.asm

You can access the macro library by using the .mlib assembler directive. The syntax for .mlib is:

.mlib [“]*filename*[“]

When the assembler encounters the .mlib directive, it opens the library named by filename and creates a table of the library's contents. The assembler enters the names of the individual members within the library into the opcode tables as library entries; this redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table.

The assembler expands the library entry in the same way it expands other macros. You can control the listing of library entry expansions with the .mlist directive. For more information about the .mlist directive, see Section 2.3. Only macros that are actually called from the library are extracted, and they are extracted only once.

You can use the archiver to create a macro library by including the desired files in an archive. A macro library is no different from any other archive, except that the assembler expects the macro library to contain macro definitions. The assembler expects only macro definitions in a macro library; putting object code or miscellaneous source files into the library may produce undesirable results.

4.5 Macro Parameters/Substitution Symbols

If you want to call a macro several times with different data each time, you can assign parameters within the macro. The macro language supports a special symbol, called a *substitution symbol*, which is used for macro parameters.

Macro parameters are substitution symbols that represent a character string. These symbols can also be used outside of macros to equate a character string to a symbol name (see Section 2.6).

Valid substitution symbols can be up to 128 characters long and *must begin with a letter*. The remainder of the symbol can be a combination of alpha- numeric characters, underscores, and dollar signs.

Substitution symbols used as macro parameters are local to the macro they are defined in. You can define up to 32 local substitution symbols (including substitution symbols defined with the .var directive) per macro. For more information about the .var directive, see Section 4.5.6.

During macro expansion, the assembler passes arguments by variable to the macro parameters. The character-string equivalent of each argument is assigned to the corresponding parameter. Parameters without corresponding arguments are set to the null string. If the number of arguments exceeds the number of parameters, the last parameter is assigned the character-string equivalent of all remaining arguments.

If you pass a list of arguments to one parameter, or if you pass a comma or semicolon to a parameter, you must surround the arguments with quotation marks.

At assembly time, the assembler replaces the macro parameter/substitution symbol with its corresponding character string, then translates the source code into object code.

Example 4-3 shows the expansion of a macro with varying numbers of arguments.

Example 4-3. Calling a Macro with Varying Numbers of Arguments

Macro definition

```
Parms  .macro a,b,c
;      a = :a:
;      b = :b:
;      c = :c: .endm
```

Calling the macro:

```
       Parms100,label             Parms100,label,x,y
;      a = 100                    ; a = 100
;      b = label                  ; b = label
;      c = " "                    ;c = x,y

Parms100, , x                     Parms"100,200,300",x,y
;      a = 100                    ; a = 100,200,300
;      b = " "                    ;b = x
;      c = x                      ;c = y

Parms"""string""",x,y
;a = "string"
;b = x
;c = y
```

4.5.1 Directives That Define Substitution Symbols

You can manipulate substitution symbols with the **.asg** and **.eval** directives.

The **.asg** directive assigns a character string to a substitution symbol. The syntax of the .asg directive is:

.asg[“]*character string*[“], *substitution symbol*

The quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the substitution symbol.

Example 4-4 shows character strings being assigned to substitution symbols.

Example 4-4. The .asg Directive

```
.asg  "IMTOREG", i2r ; Move Type
.asg  """string""", strng;string
.asg  "a,b,c", parms ; parameters
```

Example 4-4. The .asg Directive (continued)

```
mov32 { reg = A, type = IMTOREG, data = 0FFFFh } ; type = IMTOREG
```

The **.eval** directive performs arithmetic on numeric substitution symbols.

The syntax of the .eval directive is:

.eval *well-defined expresssion, substitution symbol*

The .eval directive evaluates the expression and assigns the *string value* of the result to the substitution symbol. If the expression is not well defined, the assembler generates an error and assigns the null string to the symbol.

Example 4-5 shows arithmetic being performed on substitution symbols.

Example 4-5. The .eval Directive

```
.asg  1, counter   .loop 100
.byte counter
.eval counter + 1, counter .
 endloop
```

In Example 4-5, the .asg directive could be replaced with the .eval directive (.eval 1, counter) without changing the output. In simple cases like this, you can use .eval and .asg interchangeably. However, you must use .eval if you want to calculate a value from an expression. While .asg only assigns a character string to a substitution symbol, .eval evaluates an expression and then assigns the character string equivalent to a substitution symbol.

4.5.2 Built-In Substitution Symbol Functions

The following built-in substitution symbol functions enable you to make decisions on the basis of the string value of substitution symbols. These functions always return a value, and they can be used in expressions. Built-in substitution symbol functions are especially useful in conditional assembly expressions. Parameters to these functions are substitution symbols or character-string constants.

In the function definitions shown in Table 4-6, *A* and *B* are parameters that represent substitution symbols or character-string constants. The term string refers to the string value of the parameter. The symbol *ch* represents a character constant.

Table 4-6. Functions and Return Values

Function	Return Value
$symlen(*a*)	Length of string *a*
$symcmp(*a*,*b*)	< 0 if *a* < *b* 0 if *a* = *b* > 0 if *a* > *b*
$firstch(*a*,*ch*)	Index of the first occurrence of character constant *ch* in string *a*
$lastch(*a*,*ch*)	Index of the last occurrence of character constant *ch* in string *a*
$isdefed(*a*)	1 if string *a* is defined in the symbol table 0 if string *a* is not defined in the symbol table
$ismember(*a*,*b*)	Top member of list *b* is assigned to string a 0 if *b* is a null string
$iscons(*a*)	1 if string *a* is a binary constant 2 if string *a* is an octal constant 3 if string *a* is a hexadecimal constant 4 if string *a* is a character constant 5 if string *a* is a decimal constant
$isname(*a*)	1 if string *a* is a valid symbol name 0 if string *a* is not a valid symbol name

Table 4-6. Functions and Return Values (continued)

Function	Return Value
$isreg(*a*)	1 if string *a* is a valid predefined register name 0 if string *a* is not a valid predefined register name

Example 4-6. Using Built-In Substitution Symbol Functions

```
.asg label,x                    ; x = label
.if ($symcmp(x, "label") == 0 )  ; evaluates to true
cnt { data = 0, max = 0EEEEh  }
.endif

.asg "A,B,T", list             ; list = A,B,T
.if ($ismember(x,list))        ; x = A list = B,T
mov32 { dest = start, reg = x, type = REGTOREM }  ; reg = A
.endif
```

4.5.3 Recursive Substitution Symbols

When the assembler encounters a substitution symbol, it attempts to substitute the corresponding character string. If that string is also a substitution symbol, the assembler performs substitution again. The assembler continues doing this until it encounters a token that is not a substitution symbol or until it encounters a substitution symbol that it has already encountered during this evaluation.

In Example 4-7, the A is substituted for B; B is substituted for T; and T is substituted for A. The assembler recognizes this as infinite recursion and ceases substitution.

Example 4-7. Recursive Substitution

```
.asg "A", B ; declare B and assign B = "A"
.asg "B", T ; declare T and assign T = "B"
.asg "T", A ; declare A and assign A = "T"
 mov32 { reg = A, type = IM&REGTOREG, data = 0 } ;recursive expansion
```

4.5.4 Forced Substitutions

In some cases, substitution symbols are not recognizable to the assembler. The forced substitution operator, which is a set of colons, enables you to force the substitution of a symbol's character string. Simply surround a symbol with colons to force the substitution. Do not include any spaces between the colons and the symbol.

The syntax for the forced substitution operator is:

:symbol:

The assembler expands substitution symbols surrounded by colons before expanding other substitution symbols.

You can use the forced substitution operator only inside macros, and you cannot nest a forced substitution operator within another forced substitution operator.

Example 4-8 shows how the forced substitution operator is used.

Example 4-8. Using the Forced Substitution Operator

```
force   .macro x
         .eval 0, x
```

Example 4-8. Using the Forced Substitution Operator (continued)

```
         .eval 256, y
         .eval 256, limit
         .loop 8
 AUX:x:: ecmp { reg = A, pin = CC:x:, index = x, data = y } ; The x in AUXx
         .eval x+1,x                                        ;and CCx would
         .eval limit-(x*32),y                               ;not be
         .endloop                                           ;recognizable as
         .endm                                              ;a substitution
                                                            ;symbol by the
                                                            ;assembler.
```

This would generate the following source code:

```
AUX0: ecmp { reg = A, pin = CC0, index = 0, data = 256 }
AUX1: ecmp { reg = A, pin = CC1, index = 1, data = 224 }
AUX2: ecmp { reg = A, pin = CC2, index = 2, data = 192 }
AUX3: ecmp { reg = A, pin = CC3, index = 3, data = 160 }
AUX4: ecmp { reg = A, pin = CC4, index = 4, data = 128 }
    AUX5: ecmp { reg = A, pin = CC5, index = 5, data = 96 }
AUX6: ecmp { reg = A, pin = CC6, index = 6, data = 64 }
AUX7: ecmp { reg = A, pin = CC7, index = 7, data = 32 }
```

4.5.5 Accessing Individual Characters of Subscripted Substitution Symbols

In a macro, you can access the individual characters (substrings) of a substitution symbol with subscripted substitution symbols. You must use the forced substitution operator for clarity. *The index of substring characters begins with 1, not 0.*

You can access substrings in two ways:

: symbol (*well-defined expression*):

This method of subscripting evaluates to a character string with one character.

: symbol (*well-defined expression*$_1$, *well-defined expression*$_2$):

In this method, expression1 represents the substring's starting position, and expression2 represents the substring’s length. You can specify exactly where to begin subscripting and the exact length of the resulting character string.

Example 4-9 and Example 4-10 show built-in substitution symbol functions used with subscripted substitution symbols.

Example 4-9. Using Subscripted Substitution Symbols to Redefine an Instruction

```
cmpx
         .macro x
         .var tmp
         .asg :x(1):, tmp
         .if $symcmp(tmp, "E") == 0
         .asg :x(2,$symlen(x)):, tmp
         ECMP { reg = T, data = tmp, index = 7 }
         .elseif $symcmp(tmp, "S") == 0
         .asg :x(2,$symlen(x)):, tmp
         SCMP {pin = tmp, index = 7, data = 0, action = SET }
         .elseif $symcmp(tmp, "M") == 0
         .asg :x(2,$symlen(x)):, tmp
         MCMP { reg = tmp, index = 6, data = 0, order = DATA_GE_REG }
         .else
         .emsg "Bad Macro Parameter"
         .endif
         .endm
```

Example 4-9. Using Subscripted Substitution Symbols to Redefine an Instruction (continued)

```
        cmpx   E100        ;macro call
        cmpx   SIF2        ;macro call
        cmpx   MA          ;macro call
```

In Example 4-9, the first macro call (cmpx E100) redefines the ECMP instruction and substitutes the data field by value 100. The second macro call (cmpx SIF2) redefines the SCMP instruction and substitutes the pin field by IF2. The third macro call (cmpx MA) redefines the MCMP instruction and substitutes the register field by A.

Example 4-10. Using Subscripted Substitution Symbols to Find Substrings

```
substr      macro      start,strg1,strg2,pos
            .var       len1,len2,i,tmp
            .if        $symlen(start) = 0
            .eval      1,start
            .endif
            .eval      0,pos
            .eval      start,i
            .eval      $symlen(strg1),len1
            .eval      $symlen(strg2),len2
            .loop
            .break     i = (len2 - len1 + 1)
            .asg       ":strg2(i,len1):",tmp
            .if        $symcmp(strg1,tmp) = 0
            .eval      i,pos
            .break .
            else
            .eval      i + 1,i
            .endif .
            endloop
            .endm

            .asg       0,pos
            .asg       "ar1 ar2 ar3 ar4",regs
            Substr     1,"ar2",regs,pos
            .data      pos
```

In Example 4-10, the subscripted substitution symbol is used to find a substring strg1 beginning at position start in the string strg2. The position of the substring strg1 is assigned to the substitution symbol pos.

4.5.6 Substitution Symbols as Local Variables in Macros

If you want to use substitution symbols as local variables within a macro, you can use the **.var** directive to define up to 32 local macro substitution symbols (including parameters) per macro. The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed as parameters, and they are lost after expansion.

.var sym_1 [,sym_2] ... [,sym_n]

The .var directive is used in Example 4-9 and Example 4-10.

4.6 Using Conditional Assembly in Macros

The conditional assembly directives are **.if/.elseif/.else/.endif** and **.loop/.break/.endloop**. They can be nested within each other up to 32 levels deep.

The format of a conditional block is:

.if *well-defined expression*
[**.elseif** *well-defined expression*]
[**.else**]
.endif

The **.elseif** and **.else** directives are optional in conditional assembly. The .elseif directive can be used more than once within a conditional assembly code block. When .elseif and .else are omitted and when the .if expression is false (0), the assembler continues to the code following the .endif directive.

The **.loop/.break/.endloop** directives enable you to assemble a code block repeatedly. The format of a repeatable block is:

.loop [*well-defined expression*]
[**.break** [*well-defined expression*]]
.endloop

The **.loop** directive's optional *expression* evaluates to the loop count (the number of loops to be performed). If the expression is omitted, the loop count defaults to 1024 unless the assembler encounters a .break directive with an expression that is true (nonzero).

The **.break** directive and its expression are optional in repetitive assembly. If the expression evaluates to false, the loop continues. The assembler breaks the loop when the .break expression evaluates to true or when the .break expression is omitted. When the loop is broken, the assembler continues with the code after the .endloop directive.

Example 4-11 and Example 4-12 show, respectively, .loop/.break/.endloop directives and properly nested conditional assembly directives.

Example 4-11. The .loop/.break/.endloop Directives

```
.asg 1,x
.loop

.break (x == 10)       ;if x == 10, quit loop/break
                       ;with expression
.eval x+1,x .
endloop
```

Example 4-12. Nested Conditional Assembly Directives

```
.eval1,x .loop
.if (x == 10)          ;if x == 10 quit loop
.break                 ; force break
.endif

.eval x+1,x .
endloop
```

For more information, see Section 2.5.

4.7 Using Labels in Macros

All labels in an assembly language program must be unique. This includes labels in macros. If a macro is expanded more than once, its labels are defined more than once. *Defining a label more than once is illegal.* The macro language provides a method of defining labels in macros so that the labels are unique. Simply follow the label with a question mark, and the assembler replaces the question mark with a period followed by a unique number. When the macro is expanded, *you will not see the unique number in the listing file.* Your label will appear with the question mark as it did in the macro definition. You cannot declare this label as global. The syntax for a unique label is:

*label***?**

Example 4-13. Unique Labels in a Macro

```
demo          .macro x, y, z
ecmp { next = m1?, reg = x, index = y, data = z }m1?
.endm

demo  A,   9, 100
demo  B, 0Ah, 200
demo  T, 0Bh, 300
```

The maximum label length is shortened to allow for the unique suffix. For example, if the macro is expanded fewer than 10 times, the maximum label length is 126 characters. If the macro is expanded from 10 to 99 times, the maximum label length is 125.

4.8 Producing Comments in Macros

Macro comments are comments that appear in the definition of the macro but *do not show up in the expansion of the macro.* An exclamation point in column 1 identifies a macro comment. If you want your comments to appear in the macro expansion, precede your comment with an asterisk or semicolon.

You can also produce user messages in macros by using the directives **.emsg, .mmsg**, and **.wmsg**. For more information about these directives, see Section 2.7.

Example 4-14 shows user messages in macros and macro comments that will not appear in the macro expansion.

Example 4-14. Producing Comments in a Macro

```
TEST    .MACRO    x,y !

! This macro checks for the correct number of parameters.
! The macro generates an error message if x and y are not ! present.
!
     .if ($symlen(x) == 0|$symlen(y) == 0)) ; Test for
                                            ; proper input
     .emsg “ERROR - missing parameter in call to TEST”
     .mexit
     .else
     .
     .
     .endif
     .if
     .
     .
     .endif
     .endm

   1 error, no warnings
```

4.9 Using Recursive and Nested Macros

The macro language supports recursive and nested macro calls. This means that you can call other macros in a macro definition. You can nest macros up to 32 levels deep. When you use recursive macros, you call a macro from its own definition (the macro calls itself).

When you create recursive or nested macros, you should pay close attention to the arguments that you pass to macro parameters because the assembler uses dynamic scoping for parameters. This means that the called macro uses the environment of the macro from which it was called.

Example 4-15 shows nested macros. Note that the y in the in block macro hides they in the out block macro. The x and z from the out block macro, however, are accessible to the in block macro.

Example 4-15. Using Nested Macros

```
in_block    .macro y,a

                .            ; visible parameters are y,a and
                .            ;x,z from the calling macro
                .endm

out_block   .macro x,y,z
                .            ; visible parameters are x,y,z

          in_block x,y; macro call with x and y as
                             ;arguments
                .
                . .endm
                out_block    ; macro call
```

Example 4-16 shows recursive macros. The fact macro produces assembly code necessary to calculate the factorial of n where n is an immediate value. The result is placed in the A register. The fact macro accomplishes this by calling fact1, which calls itself recursively.

Example 4-16. Using Recursive Macros

```
fact1  .macro  n

  .if n == 1
    mov32 { reg = A, data = globcnt, type = IMTOREG } ; Move immediate
                                                      ; data value to
                                                      ; register A.
  .else
    .eval n-1, temp
    .eval globcnt*temp, globcnt
    fact1 temp

  .endif
  .endm

fact  .macro n
  .if ! $iscons(n)                  ; type check input
              .emsg "Parm not a constant"
  .else
              .var temp
              .eval n, globcnt
fact1 n
  .endif
  .endm

  fact 5
```

Glossary

A

Absolute address: An address that is permanently assigned to a memory location.

Absolute lister: A debugging tool that allows you to create assembler listings that contain absolute addresses.

A_DIR: An environment variable that identifies the directory containing the commands and files necessary for running the assembler.

Allocation: A process in which the linker calculates the final memory addresses of output sections.

ALU: *Arithmetic logic unit.*

Angle value: The engine angle position in degrees. All engine functions (ignition, injection, etc.) are referenced by angle position.

Archive library: A collection of individual files that have been grouped into a single file.

Archiver: A software program that allows you to collect several individual files into a single file called an archive library. The archiver also allows you to delete, extract, or replace members of the archive library, as well as to add new members.

ASCII: *American Standard Code for Information Interchange.* A standard computer code for representing and exchanging alphanumeric information.

Assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro directives. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

Assignment statement: A statement that assigns a value to a variable.

B

Block: A set of declarations and statements that are grouped together with braces.

C

C compiler: A program that translates C source statements into assembly language source statements.

COFF: *Common object file format.* A binary object file format that promotes modular programming by supporting the concept of sections.

Command file: A file that contains linker options and names input files for the linker.

Comment: A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

Configured memory: Memory that the linker has specified for allocation.

Constant: A numeric value that can be used as an operand.

CPU: *Central processing unit.*

Cross-reference listing: An output table created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

D

Debugger: A window-oriented software interface that helps you to debug NHET programs running on an NHET simulator.

Directive: Special-purpose command that controls the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device). The NHET assembler supports two NHET-specific directives (.HETCODE and .HDA) as well as a number of generic directives.

E

Entry point: The starting execution point in target memory.

Executable module: An object file that has been linked and can be executed in a target system.

Expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

External symbol: A symbol that is used in the current program module but is defined in a different module.

Envoronmental variable: A special system symbol that the debugger uses for finding directories or obtaining debugger options.

F

File header: A portion of a COFF object file that contains general information about the object file (such as the number of section headers, the type of system the object file can be downloaded to, the number of symbols in the symbol table, and the starting address of the symbol table).

G

Global symbol: A kind of symbol that is either: 1) defined in the current module and accessed in another, or 2) accessed in the current module but defined in another.

H

NHET: *Texas Instruments Enhanced High-End Timer (NHET).* A module that provides sophisticated timing functions for complex, real-time applications, such as automobile engine management or powertrain management. These applications require the measurement of information from multiple sensors and drive actuators with complex and accurate time pulses.

High-level language debugging: The ability of a compiler to retain symbolic and high-level information (such as type and function definitions) so that a debugging tool can use this information.

L

Label: A symbol that begins in column 1 of a source statement and corresponds to the address of that statement.

Line-number entry: An entry in a COFF output module that maps lines of assembly code back to the original C source file that created them.

Linker: A software tool that combines object files to form an object module that can be allocated into target system memory and executed by the device.

Listing file: An output file, created by the assembler, that lists source statements, their line numbers, and their effects on the SPC.

Loader: A device that loads an executable module into target system memory.

M

Macro: A user-defined routine that can be used as an instruction.

Macro call: The process of invoking a macro.

Macro definition: A block of source statements that define the name and code that makes up a macro.

Macro expansion: The source statements that are substituted for the macro call and are subsequently assembled.

Macro library: An archive library composed of macros. Each file in the library must contain one macro; its name must be the same as the macro name it defines, and it must have an extension of .asm.

Memory map: A map of memory space that tells the debugger which areas of memory can and cannot be accessed.

Mnemonic: An instruction name that the assembler translates into machine code.

Module statement: Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked.

Module: An element that provides a specific function (such as a serial interface, memory, analog-to-digital conversion, timing, input / output, etc.).

O

Object file: A file that has been assembled or linked and contains machine language object code.

Object library: An archive library made up of individual object files.

Operands: The arguments, or parameters, of an assembly language instruction, assembler directive, or macro directive.

Options: Command parameters that allow you to request additional or specific functions when you invoke a software tool.

Output module: A linked, executable object file that can be downloaded and executed on a target system.

P

PC: *Personal computer.*

PCR: *Prescale capture register.*

PWM: *Pulse width Modulation.* Periodic square signal with a pulse width entity that can vary from 0 to 100%. The ratio between the pulse and the period is called the duty cycle.

R

Relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

Resolution: A value expressed in ∝s that determines the timer accuracy. All input captures, event counts, and output compares are executed once in each resolution loop. The NHET module supports resolutions from 1.0 to 12.8 ∝s.

S

Section: A relocatable block of code or data that will ultimately occupy contiguous space in the target memory map.

Section header: A portion of a COFF object file that contains information about a section in the file. Each section has its own header; the header points to the section's starting address, contains the section's size, etc.

Simulator: A software development system that simulates target system operation.

Source file: A file that contains C code or assembly language code that will be compiled or assembled to form an object file.

Source statement: The NHET assembly language source programs consist of source statements. A source statement can contain four ordered fields (label, mnemonic, operand list, and command). A single source statement can be spread over more than one line.

SPC: *Section program counter.* An element of the assembler that keeps track of the current location within a section; each section has its own SPC.

Symbol: A string of alphanumeric characters that represents an address or a value.

Symbol table: A file that contains the names of all variables in your NHET program.

Symbolic debugging: The ability of a software tool to retain symbolic information so that it can be used by a debugging tool such as a simulator or an emulator.

T

Target memory: Physical memory in a target system into which executable object codes loaded.

W

Word: A 32-bit addressable location in target memory.

Revision History

Table B-1 lists the tool changes made since the previous revision of this document.

Table B-1. Tool Revision History

User's Guide Version	Tool Version	Release Date	Author	Comments
1.4	1.4	03/05/2010	Prathap Srinivasan	Enhanced legacy NHET assembler to support –v2 option for new instructions added.
1.5	1.5	07/09/2010	Prathap Srinivasan	Enhanced tool to include std_het.h file in to the generated header file.
1.6	1.6	10/19/2010	Prathap Srinivasan	• Added assembler option –AIDx.x • Enhanced tool to accept options from input file. • DJNZ Pseudo instruction Support – Refer Note under Chapter Instruction Set.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Communications and Telecom	www.ti.com/communications
Amplifiers	amplifier.ti.com	Computers and Peripherals	www.ti.com/computers
Data Converters	dataconverter.ti.com	Consumer Electronics	www.ti.com/consumer-apps
DLP® Products	www.dlp.com	Energy and Lighting	www.ti.com/energy
DSP	dsp.ti.com	Industrial	www.ti.com/industrial
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Security	www.ti.com/security
Logic	logic.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Power Mgmt	power.ti.com	Transportation and Automotive	www.ti.com/automotive
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
		TI E2E Community Home Page	e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265