Motorola HC12 Assembler

Product	Date
HC12 Assembler	8/5/03

Table Of Contents

Assembler

This document explains how to use the Macro Assembler.

Highlights

- Graphical User Interface
- On-line Help
- 32bit Application
- Conforms to Motorola Assembly Language Input Standard

Structure of this Document

- **Graphical User Interface**: description of the Macro Assembler Graphical User Interface (GUI)
- **Environment**: detailed description of the Environment variables used by the Macro assembler
- **Assembler Options**: detailed description of the full set of Assembler options
- **Assembler Input Syntax**: detailed description of the input syntax in an assembly input file.
- **Assembler Directives**: list of all directives that the assembler supports
- **Assembler Messages**: description of messages produced by the Macro Assembler, including examples.
- **Index**

Using the Assembler

Assembler Environment

You can associate the assembler with a project directory and with an editor.

Project Directory

A project directory contains all of the environment files that you need to configure your development environment.

When you install the assembler, the assembler automatically sets the project directory to the `c:\metrowerks\demo` directory. This directory contains initialization files that are required for the tools to work correctly.

Editor

You can associate an editor with the assembler to enable the Error Feedback. You can use the *Configuration* dialog box to configure the assembler to use the editor. Please refer to the Editor Settings dialog box section of this manual.

Writing your Assembly Source File

Once your project has been configured, you can start writing your application.

Note: You can write an assembly application using one or several assembly units. Each assembly unit performs one particular task. An assembly unit is comprised of an assembly source file and some additional include files. Variables are exported and imported in the different assembly units so that a variable defined in an assembly unit can be used in another assembly unit. You create the application by linking all of the assembly units.

Let's look at an example. Suppose that your source code is in a file named `test.asm` and looks like the following code:

```
      XDEF entry   ; Make the symbol entry visible for external module.
                   ; This is necessary to allow the linker to find
                   ; the symbol and use it as the entry point for the
                   ; application.
initStk: EQU $AFE  ; Initial value for SP
dataSec: SECTION   ; Define a section
var1:    DC.W 5    ; Assign 5 to the symbol var1
```

```
codeSec: SECTION   ; Define a section for code
entry:
      LDS #initStk ; Load stack pointer
      LDD var1
      BRA entry
```

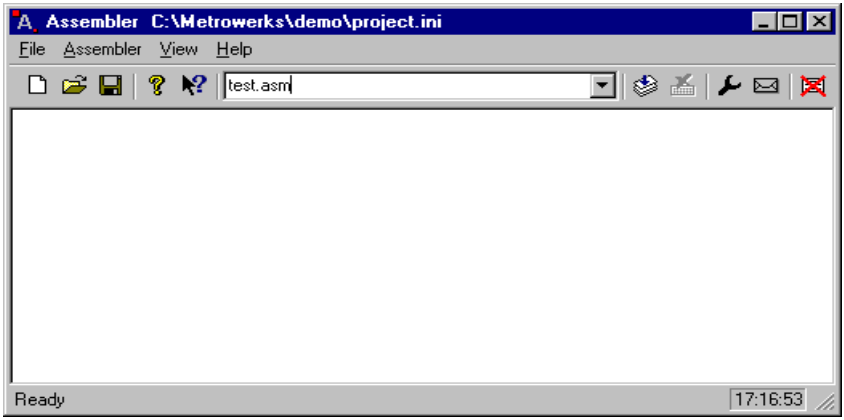
When writing your assembly source code, pay special attention to the following:

- Make sure that symbols outside of the current source file (in another source file or in the linker configuration file) that are referenced from the current source file are externally visible. Notice that we have inserted the assembly directive "`XDEF entry`" where appropriate in the example.
- In order to make debugging from the application easier, we strongly recommend that you define separate sections for code, constant data (defined with `DC`) and variables (defined with `DS`). This will mean that the symbols located in the variable or constant data sections can be displayed in the data window component.
- Make sure to initialize the stack pointer when using BSR or JSR instructions in your application.

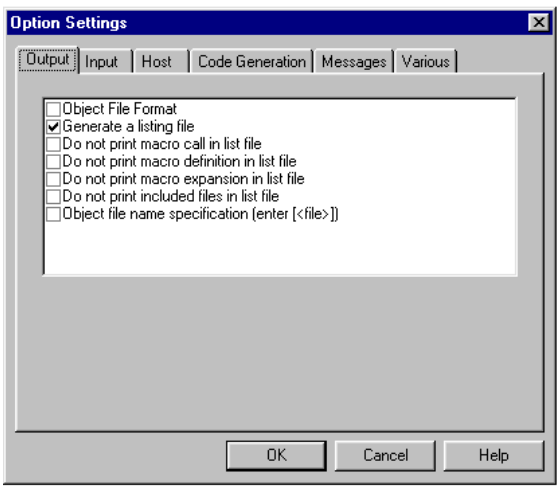
Assembling your Source File

Once the source file is available, you can assemble it.

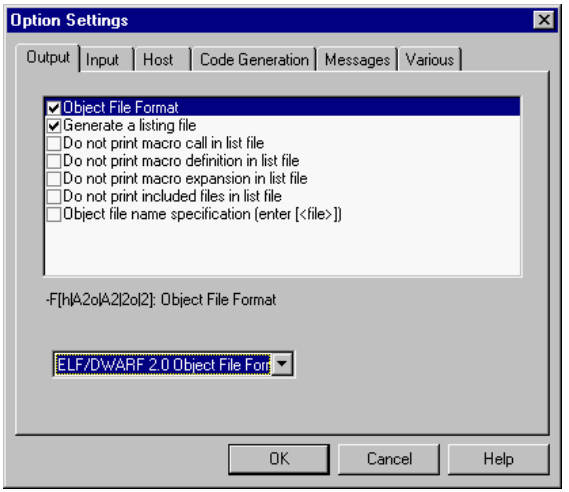
- Start the macro assembler. The *assembler* is started. You can enter the name of the file that you want to be assembled in the editable combo box. The example shows the "`test.asm`" file.



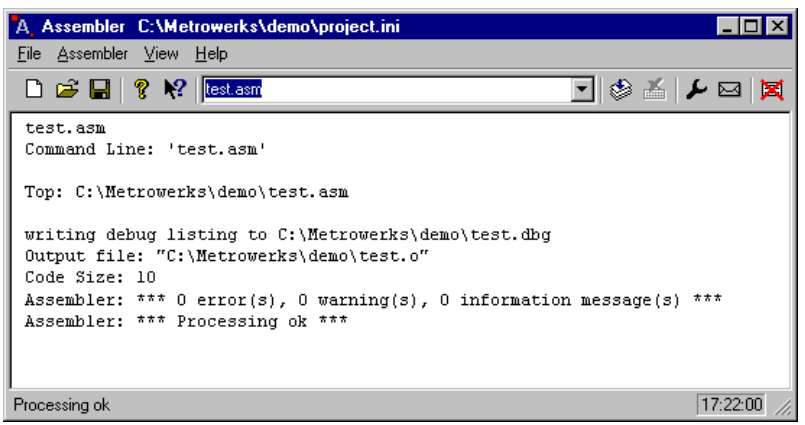
- You must correctly set the object file format (HIWARE or ELF/Dwarf). Select menu entry *Assembler | Options*. The assembler displays the *Option Settings* dialog box.



- In the *Output* folder, select the check box labeled *Object File Format*. The assembler displays more information at the bottom of the dialog box.



- Select the entry *ELF/DWARF 2.0 Object File Format* or the *HIWARE Object File Format* in the list box and click *OK*.
  The assembler starts to assemble the file when you click on the assemble button ( ).



- The Macro Assembler indicates successful assembling by printing the number of bytes of code that were generated. The message "`*** 0 error(s),`" indicates that the `test.asm` file was assembled without errors.
- The Macro Assembler generates a binary object file and a debug file for each

source file. The binary object file has the same name as the input module, but with the '`.o`' extension. The format of this file is controlled by the option -F. The debug file has the same name as the input module, but with the '`.dbg`' extension.

- When the assembly option -L is specified on the command line, the Macro Assembler generates a listing file containing the source instruction and the corresponding hexadecimal code. The listing file generated by the Macro Assembler looks like the following example:

```
HC12-Assembler
Abs. Rel.  Loc    Obj. code   Source line
---- ----  ------ ---------   -----------
  1    1                     XDEF entry ; Make the symbol entry ...
  2    2                                ; This is necessary to ...
  3    3                                ; the symbol and use it ...
  4    4                                ; application.
  5    5         0000 0AFE   initStk: EQU $AFE; Initial SP
  6    6                     dataSec: SECTION ; Define a section
  7    7  000000 0005        var1: DC.W 5     ; Assign 5 to var1
  8    8                     codeSec: SECTION ; Define a code ...
  9    9                     entry:
 10   10  000000 CF 0AFE        LDS #initStk ; Load stack pointer
 11   11  000003 FC xxxx        LDD var1
 12   12  000006 20F8           BRA entry
```

Linking Your Application

Once the object file is available, you can link your application. The linker organizes the code and data sections according to the linker parameter file.

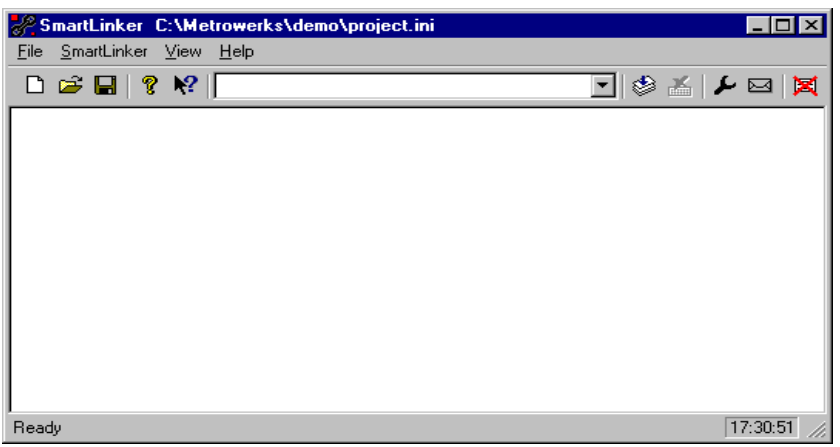
- Start your editor and create the linker parameter file. You can use the file `fibo.prm` located in the demo directory and rename it to `test.prm`.
- In the file `test.prm`, change the name of the executable and object files to test. Additionally, you can also modify the start and end address for the `ROM` and `RAM` memory area. The module `test.prm` will look like the following:

```
LINK test.abs    /* Name of the executable file generated.*/
NAMES test.o END /* Name of the object files in the application */
SECTIONS
 MY_ROM= READ_ONLY  0x800 TO 0x8FF; /* READ_ONLY memory area */
 MY_RAM= READ_WRITE 0xB00 TO 0xBFF; /* READ_WRITE memory area */
 MY_STK= READ_WRITE 0xA00 TO 0xAFF; /* READ_WRITE memory area */
END
PLACEMENT
```

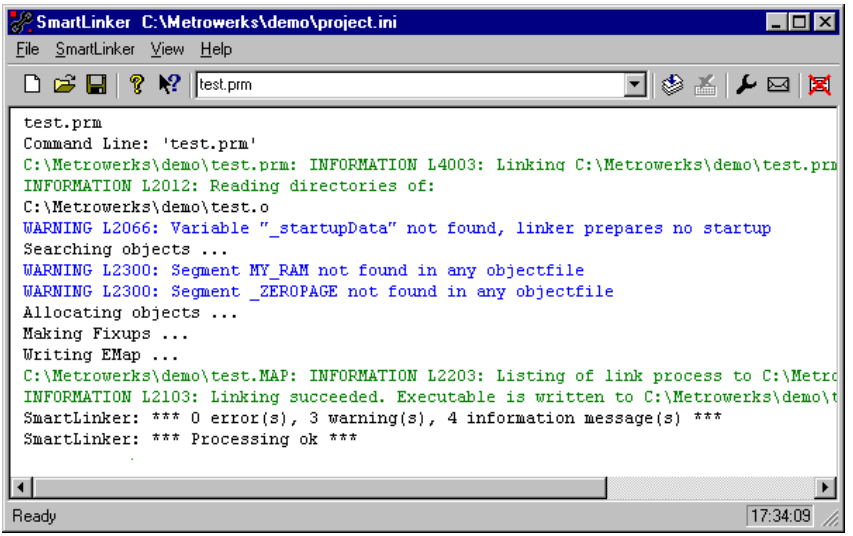
```
    DEFAULT_ROM, cstSec INTO  MY_ROM; /* Code should be
                                         allocated in MY_ROM */
    DEFAULT_RAM         INTO  MY_RAM; /* Variables should be
                                         allocated in MY_RAM */
    SSTACK              INTO MY_STK;  /* Stack will
                                        be allocated in MY_STK. */
END
INIT entry /* entry is the entry point to the application. */
VECTOR ADDRESS 0xFFFE entry
```

Note: The placement of the SSTACK section in the memory area MY_STK is optional. It is only required when the application is executed in the simulator to ensure some memory is available for the stack.

- The commands in the linker parameter file are described in detail in the linker manual.
- Start the linker.
- At the prompt, enter the name of the file that you want to link .



- Press the enter key to start linking.



Directly Generating an ABS File

The assembler can directly generate an ABS file from your assembly source file. The assembler generates a Motorola S file at the same time. You can directly burn the S file into an EPROM.

Note: The assembler for the Philips XA does not support the ELF format. Directly generating an ABS file is only possible in ELF.

Assembler source file

When an ABS file is directly generated using the assembler, no linker is involved. This means that the application must be implemented in a single assembly unit and must only contain absolute sections.

For example, suppose your source code is stored in a file named `abstest.asm` and looks like the following code:

```
        ABSENTRY entry ; Specifies the application Entry point
iniStk: EQU $AFE       ; Initial value for SP
        ORG $FFFE      ; Reset vector definition
Reset:  DC.W entry
```

```
        ORG $40         ; Define an absolute constant section
var1:   DC.B 5          ; Assign 5 to the symbol var1
        ORG $80         ; Define an absolute data section
data:   DS.B 1          ; Define one byte variable in RAM at $80
        ORG $B00        ; Define an absolute code section
entry:
        LDS  #iniStk    ; Load stack pointer
        LDAA  var1
main:
        INCA
        STAA  data
        BRA   main
```

When writing your assembly source file for direct absolute file generation, pay special attention to the following points:

- The reset vector is usually initialized in the assembly source file with the application entry point. An absolute section containing the application entry point address is created at the reset vector address. To set the entry point of the application at address `$FFFE` on the label `entry`, the following code is needed:

```
        ORG $FFFE      ; Reset vector definition
Reset:  DC.W entry
```

- The directive `ABSENTRY` is used to write the address of the application entry point in the generated absolute file. To set the entry point of the application on the label `entry` in the absolute file, the following code is needed:

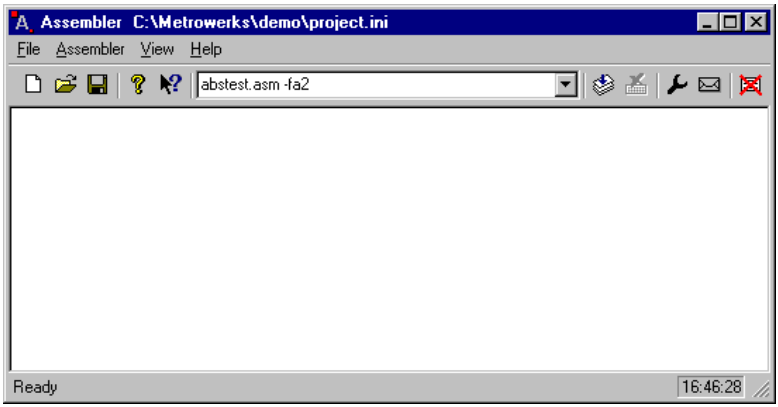
```
        ABSENTRY entry
```

- It is strongly recommended to use separate sections for code, data and constants. All sections used in the assembler application must be absolute and defined using the `ORG` directive. The address for constant or code sections has to be located in the `ROM` memory area, while the data sections have to be located in `RAM` area (according to the hardware which is used). It is the programmer's responsibility to ensure that no section overlaps occur.

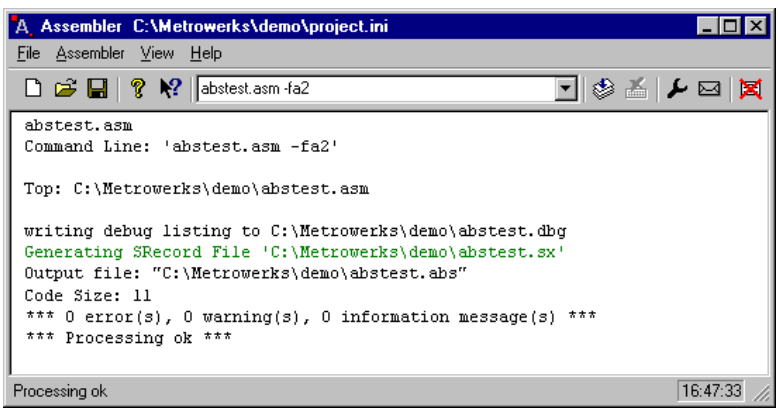
Assembling and generating the application

Once the source file is available, you can assemble it.

- Start the Macro Assembler. The assembler is started. Enter the name of the file to be assembled in the editable combo box, in our example `abstest.asm`.



- Select menu entry *Assembler* | *Options*. The *Option Settings* dialog box is displayed.
- In the Output folder, select the check box in front of the label *Object File Format*. The assembler displays more information at the bottom of the dialog box.
- Select the radio button *ELF/DWARF 2.0 Absolute File* and click *OK*. The assembler is now ready to generate directly an absolute file.
- Click the assemble button to assemble the file.



- You can load the generated absolute `.abs` file in the debugger.

- The `.sx` file generated is a standard Motorola S record file. You can directly burn this file into a EPROM memory.

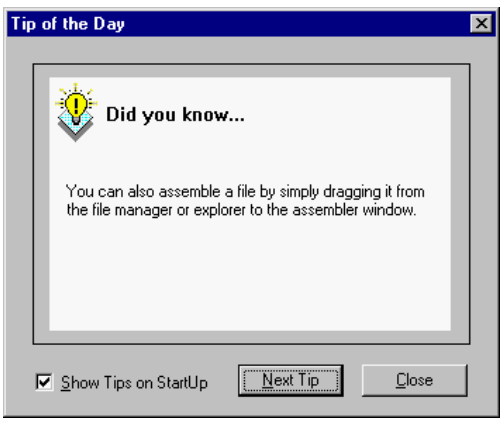
Assembler Graphical User Interface

The Macro Assembler runs under *Windows 9X, Windows NT and compatible operating systems*.

Run the assembler.

Starting the Assembler

When you start the assembler, the assembler displays a standard *Tip of the Day* window containing the news about the assembler.



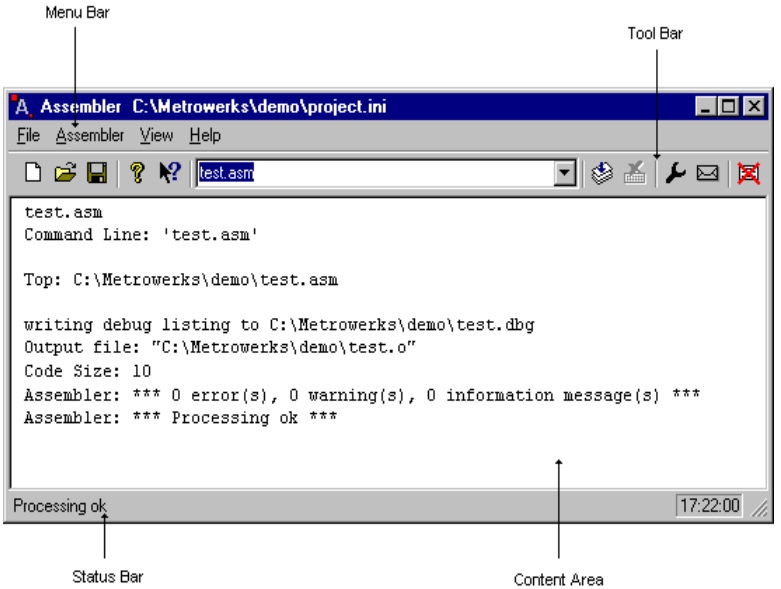
Click *Next Tip* to see the next piece of information about the assembler.

Click *Close* to close the *Tip of the Day* dialog box.

If you do not want the assembler to automatically open the standard *Tip of the Day* window when the assembler is started, uncheck *Show Tips on StartUp*.

If you want the assembler to automatically open the standard *Tip of the Day* window at assembler start up, choose *Help|Tip of the Day ...*. The assembler displays the *Tip of the Day* dialog box. Check the *Show Tips on StartUp* check box.

Assembler Main Window



This window is only visible on the screen when you do not specify any file name when you start the assembler.

The assembler window consists of a window title, a menu bar, a tool bar, a content area and a status bar.

Window Title

The window title displays the assembler name and the project name. If a project is not loaded, the assembler displays “Default Configuration” in the window title. An asterisk (*) after the configuration name indicates that some settings have changed. The assembler adds an asterisk (*) when an option, the editor configuration or the window appearance changes.

Content Area

The assembler displays logging information about the assembly session in the content area. This logging information consists of:

- the name of the file being assembled,
- the whole name (including full path specifications) of the files processed (main assembly file and all files included),
- the list of the error, warning and information messages generated and
- the size of the code generated during the assembly session.

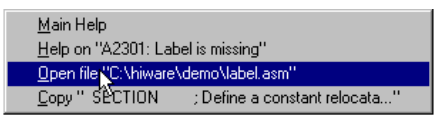
When a file is dropped into the Assembly window content area, the assembler either loads the corresponding file as a configuration file or the assembler assembles the file. The assembler loads the file as a configuration if the file has the extension `.ini`. If the file does not end with the `.ini` extension, the assembler assembles the file using the current option settings.

All text in the assembler window content area can have context information consisting of two items:

- a file name including a position inside of a file
- a message number

File context information is available for all output lines where a file name is displayed. There are two ways to open the file specified in the file context information in the editor specified in the editor configuration:

- If a file context is available for a line, double-click on a line containing file context information.
- Click with the right mouse on the line and select "*Open ..*". This entry is only available if a file context is available.

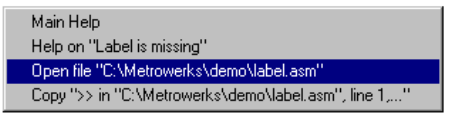


If the assembler cannot open a file even though a context menu entry is present, this means that the editor configuration information is not correct (see the section Edit Settings dialog box below).

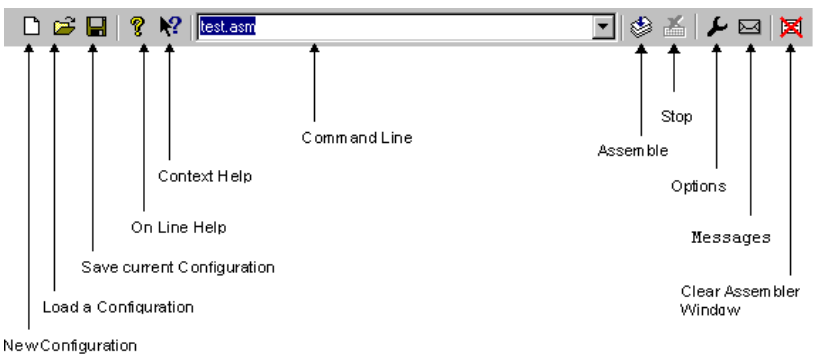
The message number is available for any message output. There are three ways to

open the corresponding entry in the help file:

- Select one line of the message and press the `F1` key. If the selected line does not have a message number, the main help is displayed.
- Press `Shift-F1` and then click on the message text. If the point clicked at does not have a message number, the main help is displayed.
- Click the right mouse button on the message text and select *Help on ....* This entry is only available if a message number is available.



Tool Bar



The three buttons on the left hand side of the toolbar correspond to the menu items of the *File* menu. The *New* , the *Load* and the *Save* buttons allow you to reset, load and save configuration files for the Macro Assembler.

The *Help* button and the *Context Help* button allow you to open the *Help* file or the *Context Help*.

When pressing , the mouse cursor changes to a question mark beside an arrow.

The assembler opens help for the next item on which you click. You can get specific help on menus, toolbar buttons or on the window area by using this *Context Help*.

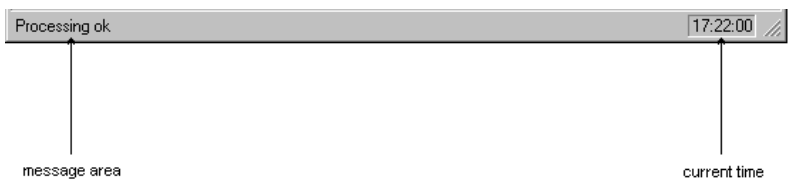
The editable combo box contains the list of the last commands executed. Once a command line has been selected or entered in this combo box, click the *Assemble* button to execute this command. The *Stop* button becomes enabled when some file is assembled. When it is pressed, the assembler stops the assembly.

The *Options Dialog Box* button allows you to open the *Option Settings* dialog.

The *Message Dialog Box* button allows you to open the *Message Settings* dialog box.

The *Clear* button allows you to clear the assembler window content area.

Status Bar



When pointing to a button in the tool bar or a menu entry, the message area displays the function of the button or menu entry you are pointing to.

Assembler Menu Bar

The following menus are available in the menu bar:

Menu	*Description*
File	*Contains entries to manage Assembler configuration files*
Assembler	*Contains entries to set Assembler options*
View	*Contains entries to customize the assembler window output*

Menu	*Description*
Help	*A standard Windows Help menu*

File Menu

With the file menu, Assembler configuration files can be saved or loaded. An Assembler configuration file contains the following information:

- the assembler option settings specified in the assembler dialog boxes
- the list of the last command line executed and the current command line.
- the window position, size and font.
- the editor currently associated with the assembler. This editor may be specifically associated with the assembler or globally defined for all *Tools* (See *Edit Settings Dialog Box*).
- the *Tips of the Day* settings, including if enabled at startup and which is the current entry.
- Configuration files are text files which have the standard extension `.ini`. The user can define as many configuration files as required for his project and can switch between the different configuration files using the *File | Load Configuration* and *File | Save Configuration* menu entry or the corresponding tool bar buttons.

Menu entry	***Description***
Assemble	*A standard Open File dialog box is opened, displaying the list of all the* `.asm` *files in the project directory. The input file can be selected using the features from the standard Open File dialog box. The selected file is assembled when the Open File dialog box is closed clicking OK*
New/Default Configuration	*Resets the assembler option settings to the default value. The assembler options which are activated per default are specified in section Assembler Options.*
Load Configuration	*A standard Open File dialog box is opened, displaying the list of all the* `.ini` *files in the project directory. The configuration file can be selected using the features from the standard Open File dialog box. The configuration data stored in the selected file is loaded and used by further assembly sessions.*

Menu entry	*Description*
Save Configuration	*Saves the current settings in the configuration file specified on the title bar.*
Save Configuration as...	*A standard Save As dialog box is opened, displaying the list of all the* `.ini` *files in the project directory. The name or location of the configuration file can be specified using the features from the standard Save As dialog box. The current settings are saved in the specified configuration file when the Save As dialog box is closed clicking OK.*
Configuration...	*Opens the Configuration dialog box to specify the editor used for error feedback and which parts to save with a configuration.* *See Editor Settings dialog box and Save Configuration dialog box*
1. .... project.ini *2. ....*	*Recent project list. This list can be used to open a recently opened project again.*
Exit	*Closes the assembler.*

Assembler Menu

This menu allows you to customize the assembler. You can graphically set or reset Assembler options or stop assembling.

Menu entry	*Description*
Options	*allows you to define the options which must be activated when assembling an input file (See Option Settings dialog box).*
Messages	*allows you to map messages to a different message class (See Messages Settings dialog box).*
Stop assembling	*Stops assembling of the current source file.*

View Menu

This menu allows you to customize the assembler window. You can specify if the status bar or the tool bar must be displayed or hidden. You can also define the font

used in the window or clear the window.

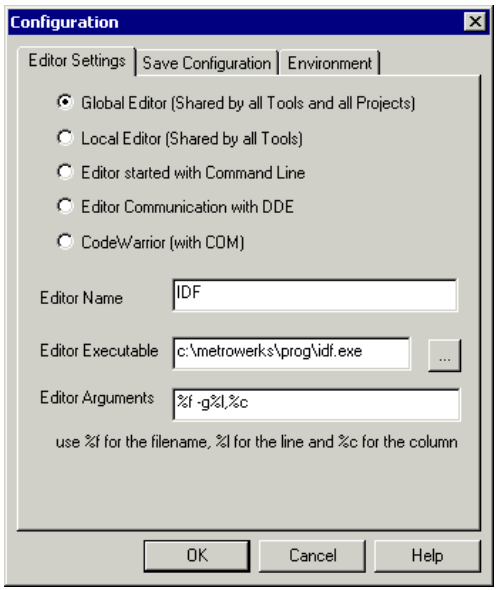
Menu entry	*Description*
Tool Bar	*switches display from the tool bar in the assembler window.*
Status Bar	*switches display from the status bar in the assembler window.*
Log...	*allows you to customize the output in the assembler window content area. The following entries are available when Log... is selected:*
Change Font	*opens a standard font dialog box. The options selected in the font dialog box are applied to the assembler window content area.*
Clear Log	*allows you to clear the assembler window content area.*

Editor Settings Dialog Box

The Editor Setting dialog box has a main selection entry. Depending on the main type of editor selected, the content below changes.

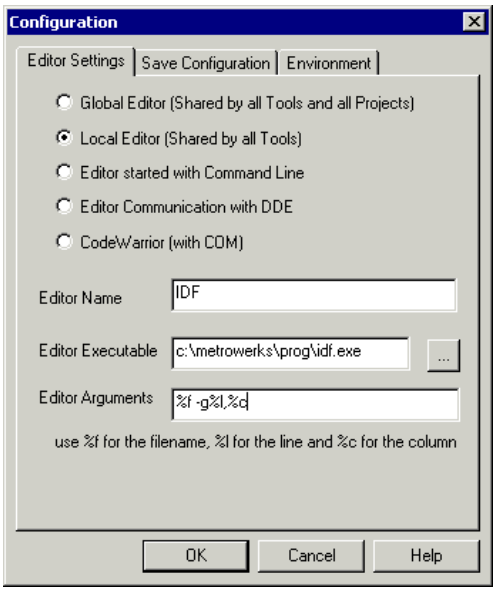
There are the following main entries:

Global Editor (Shared by all Tools and Projects)



This entry is shared by all tools (compiler/linker/assembler/...) for all projects. This setting is stored in the `[Editor]` section of the global initialization file `MCUTOOLS.INI`. Some Modifiers can be specified in the editor command line.

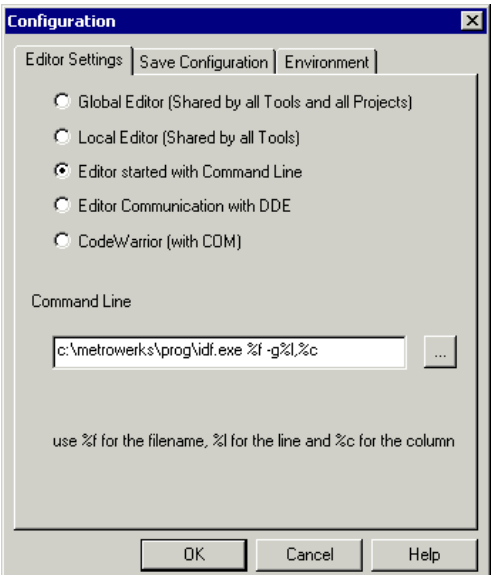
Local Editor (Shared by all Tools)



This entry is shared by all tools (compiler/linker/assembler/...) for the current project. This setting is stored in the `[Editor]` section of the local initialization file, usually `project.ini` in the current directory. Some Modifiers can be specified in the editor command line.

The global and local editor configuration affects other tools besides the assembler. It is recommended to close other tools while modifying these topics.

Editor started with Command Line



When this editor type is selected, a separate editor is associated with the assembler for error feedback. The editor configured in the shell is not used for error feedback.

Enter the command which should be used to start the editor.

The format from the editor command depends on the syntax which should be used to start the editor. Some Modifiers can be specified in the editor command line to refer to a file name of a line number (See section Modifiers below).

Example

For the *IDF* use (with an adapted path to the `idf.exe` file)

```
C:\metrowerks\prog\idf.exe %f -g%l,%c
```

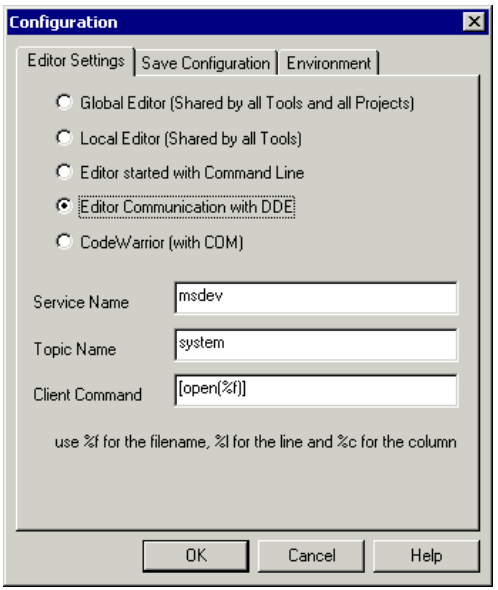
For the *CodeWright* use (with an adapted path to the `cw32.exe` file)

```
C:\cw32\cw32.exe %f -g%l
```

For *WinEdit* 32 bit version use (with an adapted path to the `winedit.exe` file)

```
C:\WinEdit32\WinEdit.exe %f /#:%l
```

Editor started with DDE



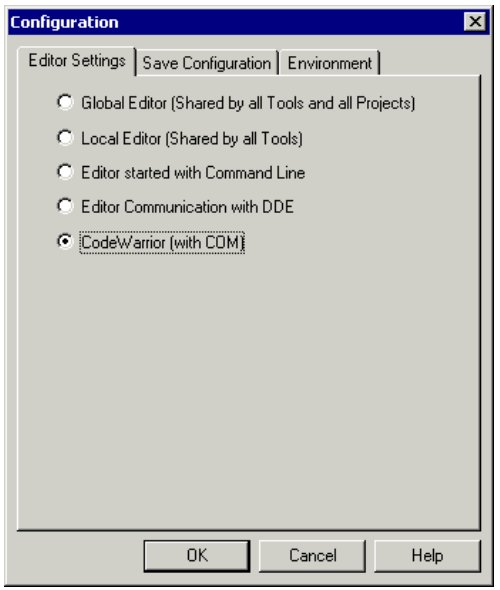
Enter the service, topic and client name to be used for a DDE connection to the editor. All entries can have modifiers for the file name and line number as explained below in the modifiers section.

Example

For Microsoft Developer Studio use the following setting:

Service Name: "msdev"
Topic Name: "system"
ClientCommand: "[open(%f)]"

CodeWarrior with COM



If CodeWarrior with COM is enabled, the CodeWarrior IDE (registered as COM server by the installation script) is used as editor.

Modifiers

The configurations may contain some modifiers to tell the editor which file to open and at which line.

- The %f modifier refers to the name of the file (including path and extension) where the error has been detected.

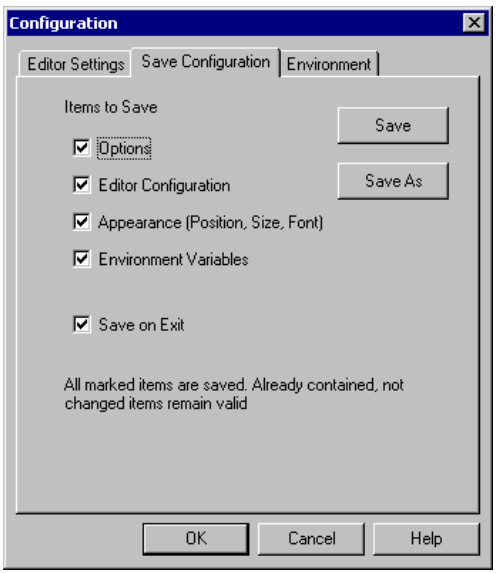
- The %l modifier refers to the line number where the message has been detected.

- The %c modifier refers to the column number where the message has been detected.

Note: Be careful. The %l modifier can only be used with an editor which can be started with a line number as parameter. This is not the case for WinEdit version 3.1 or lower or for the Notepad. When you work with such an editor,

you can start it with the file name as parameter and then select the menu entry 'Go to' to jump on the line where the message has been detected. In that case the editor command looks like:
`C:\WINAPPS\WINEDIT\Winedit.EXE %f`
Please check your editor manual to define the command line which should be used to start the editor.

Save Configuration Dialog Box



The second index of the configuration dialog box contains all options for the save operation.

In the *Save Configuration* index, four check boxes allow you to choose which items to save into a project file while the configuration is saved.

This dialog box has the following configurations:

- *Options*: This item is related to the option and message settings. If this check box is set the current option and message settings are stored in the project file when the configuration is saved. By disabling this check box, changes done to the

option and message settings are not saved, the previous settings remain valid.

- *Editor Configuration*: This item is related to the editor settings. If you set this check box, the current editor settings are stored in the project file when the configuration is saved. If you disable this check box, the previous settings remain valid.
- *Appearance*: This item is related to many parts like the window position (only loaded at startup time) and the command line content and history. If you set this check box, these settings are stored in the project file when the current configuration is saved. If you disable this check box, the previous settings remain valid.
- *Environment Variables*: With this set, the environment variable changes done in the Environment property sheet are saved too.

Note: By disabling selective options only some parts of a configuration file can be written. For example when the best Assembler options are found, the save option mark can be removed. Then future save commands will not modify the options any more.

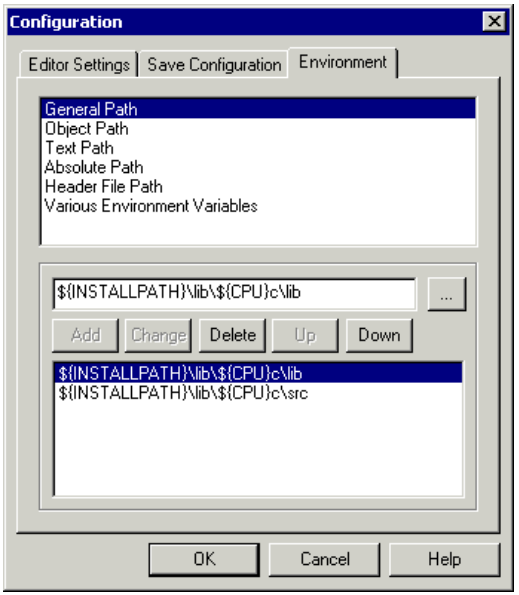
- *Save on Exit*: If this option is set, the assembler writes the configuration on exit. The assembler does not prompt you to confirm this operation. If this option is not set, the assembler does not write the configuration at exit, even if options or another part of the configuration has changed. No confirmation will appear in any case when closing the assembler.

Note: Almost all settings are stored in the project configuration file.
The only exceptions are:
- The recently used configuration list.
- All settings in this dialog box.

Note: The configurations of the assembler can, and in fact are intended to, coexist in the same file as the project configuration of other tools and the IDF. When an editor is configured by the shell, the assembler can read this content out of the project file, if present. The default project configuration file name is project.ini. The assembler does automatically open an existing project.ini in the current directory at startup. When using the option -prod at startup or loading the configuration manually, also a different name than project.ini can be chosen.

Environment Configuration Dialog

On the third page of the configuration dialog is used to configure the environment. The content of the dialog is read from the actual project file out of the section



[Environment Variables] The following variables are available:

General Path: GENPATH
Object Path: OBJPATH
Text Path: TEXTPATH
Absolute Path: ABSPATH
Header File Path: LIBPATH

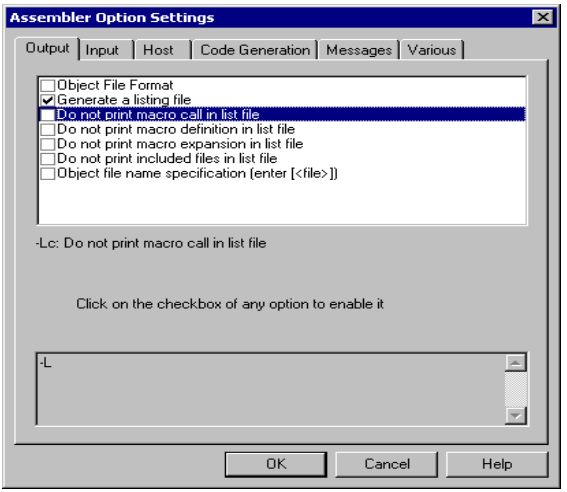
Various Environment Variables: other variables not covered by the above list.

The following buttons are available:

Add: Adds a new line/entry
Change: changes a line/entry
Delete: deletes a line/entry
Up: Moves a line/entry up
Down: Moves a line/entry down

Note that the variables are written to the project file only if you press the Save Button (or using File->Save Configuration, or CTRL-S). Additionally in the Save Configuration dialog it can be specified if the environment is written to the project file or not.

Option Settings Dialog Box



This dialog box allows you to set/reset Assembler options. The options available are arranged into different groups, and a sheet is available for each of these groups. The content of the list box depends on the selected sheet:

Group	***Description***
Output	*lists options related to the output files generation (which kind of file should be generated).*
Input	*lists options related to the input files.*
Host	*lists options related to the host.*
Code Generation	*lists options related to code generation (memory models, ...).*
Messages	*lists options controlling the generation of error messages.*
Various	*list various additional options (options used for compatibility, ...).*

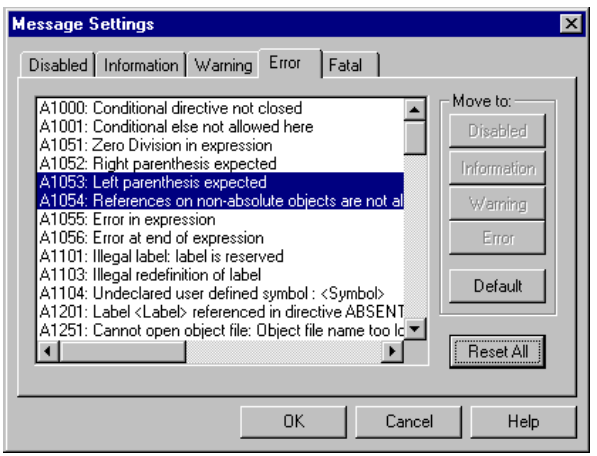
An Assembler option is set when the check box in front of it is checked. To obtain more detailed information about a specific option, select it and press the `F1` key or

the *Help* button. To select an option, click once on the option text. The option text is then displayed inverted.

When the dialog box is opened and no option is selected, pressing the `F1` key or the *Help* button shows the help about this dialog box.

The available options are listed in the section Assembler Options.

Message Settings Dialog Box



This dialog box allows you to map messages to a different message class.

Some buttons in the dialog box may be disabled, e.g. if an option cannot be moved to an information message, the ‘Move to: Information’ button is disabled. The following buttons are available in the dialog box:

Button	Description
Move to: Disabled	*The selected messages are disabled, they will no longer be displayed.*
Move to: Information	*The selected messages are changed to information messages.*
Move to: Warning	*The selected messages are changed to warning messages.*
Move to: Error	*The selected messages are changed to error messages.*

Button	Description
Move to: Default	*The selected messages are changed to their default message type.*
Reset All	*Resets all messages to their default message type.*
Ok	*Exits this dialog box and accepts the changes made.*
Cancel	*Exits this dialog box without accepting the changes made.*
Help	*Displays online help about this dialog box.*

A sheet is available for each error message class and the content of the list box depends on the selected sheet:

Message group	Description
Disabled	*Lists all messages disabled. That means that messages displayed in the list box will not be displayed by the assembler.*
Information	*Lists all information messages. Information messages informs about action taken by the assembler.*
Warning	*Lists all warning messages. When such a message is generated, translation of the input file continues and an object file will be generated.*
Error	*Lists all error messages. When such a message is generated, translation of the input file continues but no object file will be generated.*
Fatal	*Lists all fatal error messages. When such a message is generated, translation of the input file stops immediately. Fatal messages cannot be changed. They are only listed to call context help.*

Each message has its own character (‘A’ for Assembler message) followed by a 4-5 digit number. This number allows an easy search for the message both in the manual or on-line help.

Changing the Class associated with a Message

You can configure your own mapping of messages to the different classes. To do

this, use one of the buttons located on the right hand of the dialog box. Each button refers to a message class. To change the class associated with a message, you have to select the message in the list box and then click the button associated with the class where you want to move the message.

Example

To define the warning '*A2336: Value too big*' as an error message:

- Click the *Warning* sheet, to display the list of all warning messages in the list box.
- Click on the string '*A2336: Value too big*' in the list box to select the message.
- Click *Error* to define this message as an error message.

Note: Messages cannot be moved from or to the fatal error class.

Note: The 'Move to' buttons are enabled when all selected messages can be moved. When one message is marked, which cannot be moved to a specific group, the corresponding 'Move to' button is disabled (grayed).

If you want to validate the modification you have performed in the error message mapping, close the 'Message settings' dialog box with the 'OK' button. If you close it using the 'Cancel' button, the previous message mapping remains valid.

About Box

The about box can be opened with the menu Help->about. The about box contains much information including the current directory and the versions of subparts of the assembler. The main assembler version is displayed separately on top of the dialog box.

With the button 'Extended Information' it is possible to get license information about all software components in the same directory of the executable.

Click on OK to close this dialog box.

Note: During assembling, the subversions of the sub parts cannot be requested. They are only displayed if the assembler is not processing files.

Specifying the Input File

There are different ways to specify the input file which must be assembled. During assembling of a source file, the options are set according to the configuration per-

formed by the user in the different dialog boxes, and according to the options specified on the command line.

Before starting to assemble a file, make sure you have associated a working directory with your assembler.

Use the Command Line in the Tool Bar to Assemble

Assembling a New File

A new file name and additional Assembler options can be entered in the command line. The specified file is assembled when you press the *Assemble* button in the tool bar or when you press the enter key.

Assembling a file which has already been assembled

The commands executed previously can be displayed using the arrow on the right side of the command line. A command is selected by clicking on it. It appears in the command line. The specified file will be processed when the button *Assemble* in the tool bar is selected.

Use the Entry File | Assemble...

When the menu entry *File* | *Assemble...* is selected a standard file *Open File* dialog box is opened, displaying the list of all the `.asm` files in the project directory. The user can browse to get the name of the file he or she wants to assemble. Select the desired file and click *Open* in the *Open File* dialog box to assemble the selected file.

Use Drag and Drop

A file name can be dragged from an external software (for example the *File Manager*/*Explorer*) and dropped into the assembler window. The dropped file will be assembled when the mouse button is released in the assembler window. If a file being dragged has the extension `.ini`, it is considered to be a configuration and it is immediately loaded and not assembled. To assemble a source file with the extension `.ini`, use one of the other methods.

Message/Error Feedback

After assembly, there are several ways to check where different errors or warnings have been detected. Per default, the format of the error message looks as follows:

```
>> <FileName>, line <line number>, col <column number>, pos <absolute
position in file>
<Portion of code generating the problem>
<message class><message number>: <Message string>
```

Example

```
>> in "C:\metrowerks\demo\fiboerr.asm", line 18, col 0, pos 722
           DC     label
                  ^
ERROR A1104: Undeclared user defined symbol: label
```

See also Assembler options -WmsgFi, -WmsgFb, -WmsgFob, -WMsgFoi, -WmsgFonF and -WmsgFonP for different message formats.

Use Information from the Assembler Window

Once a file has been assembled, the assembler window content area displays the list of all the errors or warnings detected.

The user can use his usual editor to open the source file and correct the errors.

Use a User Defined Editor

The editor for *Error Feedback* can be configured using the *Configuration* dialog box. Error feedback is performed differently, depending on whether or not the editor can be started with a line number.

Line Number Can be Specified on the Command Line

Editors like the *IDF*, *WinEdit* (*v95* or higher) or *Codewright* can be started with a line number in the command line. When these editors have been correctly configured, they can be started automatically by double clicking on an error message. The configured editor will be started, the file where the error occurs is automatically opened and the cursor is placed on the line where the error was detected.

Line Number Cannot be Specified on The Command Line

Editors like *WinEdit v31* or lower, *Notepad*, *Wordpad* cannot be started with a line number in the command line. When these editors have been correctly configured, they can be started automatically by double clicking on an error message. The configured editor will be started and the file where the error occurs is automatically opened. To scroll to the position where the error was detected, you have to:

- Activate the assembler again.

- Click the line on which the message was generated. This line is highlighted on the screen.
- Copy the line in the clipboard pressing `CTRL + C`.
- Activate the editor again.
- Select *Search* | *Find*, the standard *Find* dialog box is opened.
- Copy the content of the clipboard in the Edit box pressing `CTRL + V`.
- Click *Forward* to jump to the position where the error was detected.

Environment

This part describes the environment variables used by the assembler. Some of those environment variables are also used by other tools (e.g. Linker/Compiler), so consult also their respective manual.

There are three ways to specify of environment:

1) The current project file with the section [Environment Variables]. This file may be specified on Tool startup using the -Prod option. This way is recommended and also supported by the IDF.

2) An optional 'default.env' file in the current directory. This file is supported for compatibility reasons with earlier versions. The name of this file may be specified using the variable ENVIRONMENT. Using the default.env file is not recommended.

3) Setting environment variables on system level (DOS level). This is not recommended.

Various parameters of the assembler may be set in an environment using so-called environment variables. The syntax is always the same:

```
Parameter = KeyName "=" ParamDef.
```

Example

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/lib;/home/me/
my_project
```

These parameters may be defined in several ways:

Using system environment variables supported by your operating system.

Putting the definitions in a file called `DEFAULT.ENV` (`.hidefaults` for UNIX) in the default directory.

Putting the definitions in a file given by the value of the system environment variable ENVIRONMENT.

Note: The default directory mentioned above can be set via the system environment variable `DEFAULTDIR`

When looking for an environment variable, all programs first search the system environment, then the `DEFAULT.ENV` (`.hidefaults` for UNIX) file and finally the global environment file given by ENVIRONMENT. If no definition can be found, a default value is assumed.

Note: The environment may also be changed using the -Env Assembler option

The Current Directory

The most important environment for all tools is the current directory. The current directory is the base search directory where the tool starts to search for files (e.g. for the DEFAULT.ENV / .hidefaults)

Normally, the current directory of a tool started is determined by the operating system or by the program who launches the tools (e.g. IDF, Make Utility, ...).

For the UNIX operating system, the current directory for an executable is also the current directory from where the binary file has been started.

For MS Windows based operating systems, the current directory definition is quite complex:

- If the tool is launched using a File Manager/Explorer, the current directory is the location of the executable launched.
- If the tool is launched using an Icon on the Desktop, the current directory is the one specified and associated with the Icon in its properties.
- If the tool is launched by dragging a file on the icon of the executable under Windows 95 or Windows NT 4.0, the desktop is the current directory.
- If the tool is launched by another launching tool with its own current directory specification (e.g. an editor as IDF, a Make utility, ...), the current directory is the one specified by the launching tool (e.g. current directory definition in IDF).
- When local project file is loaded, the current directory is set to where the local project file is in. Changing the current project file does also change the current directory if the other project file is in a different directory. Note that browsing for an assembly source file does not change the current directory.

To overwrite this behavior, the system environment variable DEFAULTDIR may be used.

The current directory is displayed among other information with the assembler option “-v” and in the about box.

Environment Macros

It is possible to use Macros in your environment settings.

Example:

```
MyVAR=C:\test
TEXTPATH=$(MyVAR)\txt
OBJPATH=${MyVAR}\obj
```

In the example, TEXTPATH is expanded to ‘C:\test\txt’ and OBJPATH is expanded to ‘C:\test\obj’.

From the example above, you can see that you either can use $() or ${}. However, the variable referenced has to be defined somewhere.

Additionally there are following special variables allowed too (note that the are always surrounded by {} and they are case sensitive. Additionally the variable content contains a the directory separator ‘\’ as well:

- {Compiler}: That is the path of the executable one directory level up. That is if the executable is ‘c:\metrowerks\prog\linker.exe’, then the variable is ‘c:\metrowerks\’. Note that {Compiler} is used for the assembler too.
- {Project}: Path of the current project file. E.g. if the current project file is ‘C:\demo\project.ini’, the variable contains ‘C:\demo\’.
- {System}: This is the path were your Windows system is installed, e.g. ‘C:\WINNT\’.

Global Initialization File (MCUTOOLS.INI) (PC only)

All tools may store some global data into the file `MCUTOOLS.INI`. The tool first search for this file in the directory of the tool itself (path of the executable). If there is no `MCUTOOLS.INI` file in this directory, the tool looks for a `MCUTOOLS.INI` file located in the *MS Windows* installation directory (e.g. `C:\WINDOWS`).

Example

```
C:\WINDOWS\MCUTOOLS.INI
D:\INSTALL\PROG\MCUTOOLS.INI
```

If a tool is started in the `D:\INSTALL\PROG\DIRECTOY`, the initialization file in the same directory than the tool is used (`D:\INSTALL\PROG\MCUTOOLS.INI`).

But if the tool is started outside the `D:\INSTALL\PROG` directory, the initialization file in the *Windows* directory is used (`C:\WINDOWS\MCUTOOLS.INI`).

The following section gives a short description of the entries in the `MCUTOOLS.INI` file:

[Installation] Section

Entry: Path

Arguments: Last installation path.

Description Whenever a tool is installed, the installation script stores the installation destination directory into this variable.

Example `Path=c:\install`

Entry: Group

Arguments: Last installation program group.

Description Whenever a tool is installed, the installation script stores the installation program group created into this variable.

Example `Group=Assembler`

[Options] Section

Entry: DefaultDir

Arguments: Default Directory to be used.

Description Specifies the current directory for all tools on a global level (see also environment variable DEFAULTDIR).

Example `DefaultDir=c:\install\project`

[XXX_Assembler] Section

instead of XXX, the actual backend name appears

Entry: SaveOnExit

Arguments: 1/0

Description 1 if the configuration should be stored when the assembler is closed, 0 if it should not be stored. The assembler does not ask to store a configuration in either cases.

Entry: SaveAppearance

Arguments: 1/0

Description 1 if the visible topics should be stored when writing a project file, 0 if not. The command line, its history, the windows position and other topics belong to this entry.
This entry corresponds to the state of the check box ‘Appearance’ in the ‘Save Configuration’ dialog box.

Entry: SaveEditor

Arguments: 1/0

Description 1 if the editor settings should be stored when writing a project file, 0 if not. The editor setting contain all information of the editor configuration dialog box.
This entry corresponds to the state of the check box ‘Editor Configuration’ in the ‘Save Configuration’ dialog box.

Entry: SaveOptions

Arguments: 1/0

Description 1 if the options should be contained when writing a project file, 0 if not.
This entry corresponds to the state of the check box ‘Options’ in the ‘Save Configuration’ dialog box.

Entry: RecentProject0, RecentProject1, ...

Arguments: names of the last and prior project files

Description This list is updated when a project is loaded or saved. Its current content is shown in the file menu.

Example
```
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=C:\myprj\project.ini
RecentProject1=C:\otherprj\project.ini
```

[Editor] Section

Entry: Editor_Name

Arguments: The name of the global editor

Description: Specifies the name of the editor used as global editor. This entry has only a description effect. Its content is not used to start the editor.

Saved: Only with ‘Editor Configuration’ set in the File->Configuration Save Configuration dialog box.

Entry: Editor_Exe

Arguments: The name of the executable file of the global editor (including path).

Description: Specifies the file name which is started to edit a text file, when the global editor setting is active.

Saved: Only with ‘Editor Configuration’ set in the File->Configuration Save Configuration dialog box.

Entry: Editor_Opts

Arguments: The options to use with the global editor

Description: Specifies options (arguments), which should be used when starting the global editor. If this entry is not present or empty, “%f” is used. The command line to launch the editor is build by taking the Editor_Exe content, then appending a space followed by the content of this entry.

Saved: Only with ‘Editor Configuration’ set in the File->Configuration Save Configuration dialog box.

Example

```
[Editor]
editor_name=IDF
editor_exe=C:\metrowerks\prog\idf.exe
editor_opts=%f -g%l,%c
```

Example

The following example shows a typical layout of the `MCUTOOLS.INI`:

```
[Installation]
Path=c:\metrowerks
Group=Assembler

[Editor]
editor_name=IDF
editor_exe=C:\metrowerks\prog\idf.exe
editor_opts=%f -g%l,%c

[Options]
DefaultDir=c:\myprj

[XXX_Assembler]
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=c:\myprj\project.ini
RecentProject1=c:\otherprj\project.ini
```

Local Configuration File (usually project.ini)

The assembler does not change the default.env file in any way. The assembler only reads the contents. All the configuration properties are stored in the configuration file. The same configuration file can and is intended to be used by different applications (assembler, Linker, etc.).

The processor name is encoded into the section name, so that Assembler for different processors can use the same file without any overlapping. Different versions of the same Assembler are using the same entries. This mainly plays a role when options only available in one version should be stored in the configuration file. In such situations, two files must be maintained for the different Assembler versions. If no incompatible options are enabled when the file is last saved, the same file can be used for both Assembler version.

The current directory is always the directory, where the configuration is in. If a configuration file in a different directory is loaded, then the current directory also changes. When the current directory changes, also the whole default.env file is reloaded. Always when a configuration file is loaded or stored, the options in the environment variable `ASMOPTIONS` is reloaded and added to the project options. This behavior has to be noticed when in different directories different default.env

exist which do contain incompatible options in `ASMOPTIONS`. When a project is loaded using the first default.env, its `ASMOPTIONS` are added to the configuration file. If then this configuration is stored in a different directory, where a default.env exists with the incompatible options, the assembler adds options and remarks the inconsistency. Then a message box appears to inform the user that the default.env options were not added. In such a situation the user can either remove the option from the configuration file with the advanced option dialog box or he can remove the option from the default.env with the shell or a text editor depending which options should be used in the future.

At startup the configuration stored in the file `project.ini` located in the current directory is loaded.

[Editor] Section

Entry:	Editor_Name
Arguments:	The name of the local editor
Description	Specifies the name of the editor used as local editor. This entry has only a description effect. Its content is not used to start the editor. This entry has the same format as for the global editor configuration in the `mcutools.ini` file.
Saved:	Only with ‘Editor Configuration’ set in the File->Configuration Save Configuration dialog box.

Entry:	Editor_Exe
Arguments:	The name of the executable file of the local editor (including path).
Description	Specifies file name with is started to edit a text file, when the local editor setting is active. In the editor configuration dialog box, the local editor selection is only active when this entry is present and not empty. This entry has the same format as for the global editor configuration in the `mcutools.ini` file.
Saved:	Only with ‘Editor Configuration’ set in the File->Configuration Save Configuration dialog box.

Entry:	Editor_Opts
Arguments:	The options to use with the local editor
Description	Specifies options (arguments), which should be used when starting the local editor. If this entry is not present or empty, "%f" is used. The command line to launch the editor is build by taking the Editor_Exe content, then appending a space followed by the content of this entry. This entry has the same format as for the global editor configuration in the `mcutools.ini` file.
Saved:	Only with 'Editor Configuration' set in the File->Configuration Save Configuration dialog box.

Example

```
[Editor]
editor_name=IDF
editor_exe=C:\metrowerks\prog\idf.exe
editor_opts=%f -g%l,%c
```

[XXX_Assembler] Section

instead of XXX, the actual backend name appears

Entry:	RecentCommandLineX, X= integer
Arguments:	String with a command line history entry, e.g. `fibo.asm`
Description	This list of entries contains the content of the command line history.
Saved:	Only with Appearance set in the File->Configuration Save Configuration dialog box.

Entry:	CurrentCommandLine
Arguments:	String with the command line, e.g. "fibo.asm -w1"
Description	The currently visible command line content.
Saved:	Only with Appearance set in the File->Configuration Save Configuration dialog box.

Entry:	StatusbarEnabled
Arguments:	1/0
Special:	This entry is only considered at startup. Later load operations do not use it any more.
Description	Is currently the statusbar enabled state. 1: the statusbar is visible 0: the statusbar is hidden
Saved:	Only with Appearance set in the File->Configuration Save Configuration dialog box.

Entry:	ToolbarEnabled
Arguments:	1/0
Special:	This entry is only considered at startup. Later load operations do not use it any more.
Description	Is currently the toolbar enabled state. 1: the toolbar is visible 0: the toolbar is hidden
Saved:	Only with Appearance set in the File->Configuration Save Configuration dialog box.

Entry:	WindowPos
Arguments:	10 integers, e.g. “0,1,-1,-1,-1,-1,390,107,1103,643”
Special:	This entry is only considered at startup. Later load operations do not use it any more. Changes of this entry do not show the “*” in the title.
Description	This numbers contain the position and the state of the window (maximized,..) and other flags.
Saved:	Only with Appearance set in the File->Configuration Save Configuration dialog box.

Entry:	WindowFont
Arguments:	size: == 0 -> generic size, < 0 -> font character height, > 0 font cell height weight: 400 = normal, 700 = bold (valid values are 0..1000) italic: 0 == no, 1 == yes font name: max. 32 characters.
Description	Font attributes.
Saved:	Only with Appearance set in the File->Configuration Save Configuration dialog box.
Example	`WindowFont=-16,500,0,Courier`

Entry:	TipFilePos
Arguments:	any integer, e.g. 236
Description	Actual position in tip of the day file. Used that different tips are shown at different calls.
Saved:	Always when saving a configuration file.

Entry:	ShowTipOfDay
Arguments:	0/1
Description	Should the Tip of the Day dialog box be shown at startup. 1: it should be shown 0: no, only when opened in the help menu
Saved:	Always when saving a configuration file.

Entry:	Options
Arguments:	current option string, e.g.: -W2
Description	The currently active option string. This entry can be very long.
Saved:	Only with Options set in the File->Configuration Save Configuration dialog box.

Entry:	EditorType
Arguments:	0/1/2/3/4
Description	This entry specifies which editor configuration is active. 0: global editor configuration (in the file mcutools.ini) 1: local editor configuration (the one in this file) 2: command line editor configuration, entry EditorCommandLine 3: DDE editor configuration, entries beginning with EditorDDE 4: CodeWarrior with COM. There are no additional entries. For details see also Editor Setting dialog box.
Saved:	Only with Editor Configuration set in the File->Configuration Save Configuration dialog box.

Entry:	EditorCommandLine
Arguments:	command line, for IDF: “c:\metrowerks\prog\idf.exe %f -g%l,%c”
Description	Command line content to open a file. For details see also Editor Setting dialog box.
Saved:	Only with Editor Configuration set in the File->Configuration Save Configuration dialog box.

Entry:	EditorDDEClientName
Arguments:	client commend, e.g. “[open(%f)]”
Description	Name of the client for DDE editor configuration. For details see also Editor Setting dialog box.
Saved:	Only with Editor Configuration set in the File->Configuration Save Configuration dialog box.

Entry:	EditorDDETopicName
Arguments:	topic name, e.g. “system”
Description	Name of the topic for DDE editor configuration. For details see also Editor Setting dialog box.

Saved:	Only with Editor Configuration set in the File->Configuration Save Configuration dialog box.

Entry:	EditorDDEServiceName
Arguments:	service name, e.g. “system”
Description	Name of the service for DDE editor configuration. For details see also Editor Setting dialog box.
Saved:	Only with Editor Configuration set in the File->Configuration Save Configuration dialog box.

Example

The following example shows a typical layout of the configuration file (usually project.ini):

```
[Editor]
Editor_Name=IDF
Editor_Exe=c:\metrowerks\prog\idf.exe
Editor_Opts=%f -g%l,%c

[XXX_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
TipFilePos=0
ShowTipOfDay=1
Options=-w1
EditorType=3
RecentCommandLine0=fibo.asm -w2
RecentCommandLine1=fibo.asm
CurrentCommandLine=fibo.asm -w2
EditorDDEClientName=[open(%f)]
EditorDDETopicName=system
EditorDDEServiceName=msdev
EditorCommandLine=c:\metrowerks\prog\idf.exe %f -g%l,%c
```

Paths

Most environment variables contain path lists telling where to look for files. A path

list is a list of directory names separated by semicolons following the syntax below:

```
PathList = DirSpec {";" DirSpec}.
DirSpec  = ["*"] DirectoryName.
```

Example

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/metrowerks/lib;/
home/me/my_project
```

If a directory name is preceded by an asterisk (`"*"`), the programs recursively search that whole directory tree for a file, not just the given directory itself. The directories are searched in the order they appear in the path list.

Example

```
LIBPATH=*C:\INSTALL\LIB
```

Note: Some DOS/UNIX environment variables (like GENPATH, LIBPATH, etc.) are used. For further details refer to Environment Variable Details.

We strongly recommend working with the Shell and setting the environment by means of a `DEFAULT.ENV` file in your project directory (This 'project dir.' can be set in the Shell's 'Configure' dialog box). This way, you can have different projects in different directories, each with its own environment

Note: When starting the assembler from an external editor, do not set the system environment variable `DEFAULTDIR`. If you do so and this variable does not contain the project directory given in the editor's project configuration, files might not be put where you expect them to be put!

Fore some environment variables a synonym also exists. Those synonyms may be used for older releases of the assembler and will be removed in the future.

Line Continuation

It is possible to specify an environment variable in a environment file (default.env/ .hidefaults) over different lines using the line continuation character '\':

Example

```
ASMOPTIONS=\
-W2 \
-WmsgNe=10
```

This is the same as

```
ASMOPTIONS=-W2 -WmsgNe=10
```

But this feature may be dangerous using it together with paths, e.g.

```
GENPATH=.\
TEXTFILE=.\txt
```

will result in

```
GENPATH=.TEXTFILE=.\txt
```

To avoid such problems, we recommend to use a semicolon';' at the end of a path if there is a '\' at the end:

```
GENPATH=.\;
TEXTFILE=.\txt
```

Environment Variable Details

The remainder of this section is devoted to describing each of the environment variables available for the assembler. The environment variables are listed in alphabetical order and each is divided into several sections.

Topic	***Description***
Tools	*Lists tools which are using this variable.*
Synonym	*Fore some environment variables a synonym also exists. Those synonyms may be used for older releases of the assembler and will be removed in the future. A synonym has lower precedence than the environment variable.*
Syntax	*Specifies the syntax of the option in a EBNF format.*
Arguments	*Describes and lists optional and required arguments for the variable.*
Default	*Shows the default setting for the variable or none.*
Description	*Provides a detailed description of the option and how to use it.*
Example	*Gives an example of usage, and effects of the variable where possible. The examples shows an entry in the* `default.env` *for PC or in the* `.hidefaults` *for UNIX.*
See also	*Names related sections.*

ABSPATH

ABSPATH: Absolute file Path

Tools:	Compiler, Assembler, Linker, Decoder, Debugger
Synonym:	None
Syntax:	"ABSPATH=" {<path>}.
Arguments:	<path>: Paths separated by semicolons, without spaces.
Default:	none.
Description	This environment variable is only relevant when absolute files are directly generated by the macro assembler instead of object files. When this environment variable is defined, the assembler will store the absolute files it produces in the first directory specified there. If ABSPATH is not set, the generated absolute files will be stored in the directory the source file was found.
Example	ABSPATH=\sources\bin;..\..\headers;\usr\local\bin
See also	none

ASMOPTIONS

ASMOPTIONS: Default Assembler Options

Tools:	Assembler
Synonym:	None
Syntax:	"ASMOPTIONS=" {<option>}.
Arguments:	<option>: Assembler command line option
Default:	none.
Description	If this environment variable is set, the assembler appends its contents to its command line each time a file is assembled. It can be used to globally specify certain options that should always be set, so you don't have to specify them each time a file is assembled. Options enumerated there must be valid assembly options and are separated by space characters.
Example	ASMOPTIONS=-W2 -L
See also	Assembler options

COPYRIGHT

COPYRIGHT: Copyright Entry in Object File

Tools:	Compiler, Assembler, Linker, Librarian
Synonym:	none.
Syntax:	"COPYRIGHT=" <copyright>.
Arguments:	<copyright>: copyright entry.
Default:	none.
Description	Each object file contains an entry for a copyright string. This information may be retrieved from the object files using the decoder.
Example	COPYRIGHT=Copyright
See also	Environment variable USERNAME
	Environment variable INCLUDETIME

DEFAULTDIR

DEFAULTDIR: Default Current Directory

Tools:	Compiler, Assembler, Linker, Decoder, Debugger, Librarian, Maker
Synonym:	none.
Syntax:	"DEFAULTDIR=" <directory>.
Arguments:	<directory>: Directory to be the default current directory.
Default:	none.
Description	With this environment variable the default directory for all tools may be specified. All the tools indicated above will take the directory specified as their current directory instead the one defined by the operating system or launching tool (e.g. editor). *Note: This is an environment variable on system level (global environment variable) It cannot be specified in a default environment file (DEFAULT.ENV/.hidefaults)*
Example	DEFAULTDIR=C:\INSTALL\PROJECT
See also	Section The Current Directory Section Global Initialization File (MCUTOOLS.INI)

ENVIRONMENT

ENVIRONMENT: Environment File Specification

Tools:	Compiler, Assembler, Linker, Decoder, Debugger, Librarian, Maker
Synonym:	HIENVIRONMENT
Syntax:	"ENVIRONMENT=" <file>.
Arguments:	<file>: file name with path specification, without spaces
Default:	none.
Description	This variable has to be specified on system level. Normally the assembler looks in the current directory for a environment file named default.env (.hidefaults on UNIX). Using ENVIRONMENT (e.g. set in the autoexec.bat (DOS) or .cshrc (UNIX)), a different file name may be specified. *Note: This is an environment variable on system level (global environment variable) It cannot be specified in a default environment file (DEFAULT.ENV/.hidefaults).*
Example	ENVIRONMENT=\metrowerks\prog\global.env
See also	none.

ERRORFILE

ERRORFILE: Error File Name Specification

Tools:	Compiler, Assembler, Linker
Synonym:	none.
Syntax:	"ERRORFILE=" <file name>.
Arguments:	<file name>: File name with possible format specifiers.
Default:	EDOUT.
Description	The environment variable ERRORFILE specifies the name for the error file (used by the Compiler or assembler). Possible format specifiers are: '%n': Substitute with the file name, without the path. '%p': Substitute with the path of the source file. '%f': Substitute with the full file name, i.e. with the path and name (the same as '%p%n'). In case of an illegal error file name, a notification box is shown.

Example	`ERRORFILE=MyErrors.err`
	
	lists all errors into the file MyErrors.err in the current directory.
	
	`ERRORFILE=\tmp\errors`
	
	lists all errors into the file errors in the directory \tmp.
	
	`ERRORFILE=%f.err`
	
	lists all errors into a file with the same name as the source file, but with extension .err, into the same directory as the source file, e.g. if we compile a file \sources\test.c, an error list file \sources\test.err will be generated.
	
	`ERRORFILE=\dir1\%n.err`
	
	for a source file test.c, an error list file \dir1\test.err will be generated.

```
ERRORFILE=%p\errors.txt
```

for a source file \dir1\dir2\test.c, an error list file \dir1\dir2\errors.txt will be generated.

If the environment variable ERRORFILE is not set, errors are written to the default error file. The default error file name depends on the way the assembler is started.

If a file name is provided on the assembler command line, the errors are written to the file EDOUT in the project directory.

If no file name is provided on the assembler command line, the errors are written to the file ERR.TXT in the project directory.

Example

Another example shows the usage of this variable to support correct error feedback with the WinEdit Editor which looks for an error file called EDOUT:

```
Installation directory: E:\INSTALL\PROG
Project sources: D:\SRC
Common Sources for projects: E:\CLIB

Entry in default.env (D:\SRC\DEFAULT.ENV):
ERRORFILE=E:\INSTALL\PROG\EDOUT

Entry in WINEDIT.INI (in Windows directory):
OUTPUT=E:\INSTALL\PROG\EDOUT
```

Note: Be careful to set this variable if the WinEdit Editor is use, else the editor cannot find the EDOUT file

See also

none.

GENPATH

GENPATH: Search Path for Input File

Tools:	Compiler, Assembler, Linker, Decoder, Debugger
Synonym:	HIPATH
Syntax:	"GENPATH=" {<path>}.
Arguments:	<path>: Paths separated by semicolons, without spaces.
Default:	none.
Description	The macro assembler will look for the sources and included files first in the project directory, then in the directories listed in the environment variable GENPATH
	Note: If a directory specification in this environment variables starts with an asterisk (“”), the whole directory tree is searched recursive depth first, i.e. all subdirectories and their subdirectories and so on are searched, too. Within one level in the tree, search order of the subdirectories is indeterminate (these is not valid for Win32).*
Example	GENPATH=\sources\include;..\..\headers;\usr\local\lib
See also	none.

INCLUDETIME

INCLUDETIME: Creation Time in Object File

Tools:	Compiler, Assembler, Linker, Librarian
Synonym:	none.
Syntax:	"INCLUDETIME=" ("ON" | "OFF").
Arguments:	"ON": Include time information into object file.
	"OFF": Do not include time information into object file.
Default:	"ON"
Description	Normally each object file created contains a time stamp indicating the creation time and data as strings. So whenever a new file is created by one of the tools, the new file gets a new time stamp entry.
	This behavior may be undesired if for SQA reasons a binary file compare has to be performed. Even if the information in two object files is the same, the files do not match exactly because the time stamps are not the same. To avoid such problems this variable may be set to OFF. In this case the time stamp strings in the object file for date and time are “none” in the object file.
	The time stamp may be retrieved from the object files using the decoder.
Example	INCLUDETIME=OFF
See also	Environment variable COPYRIGHT
	Environment variable USERNAME

OBJPATH

OBJPATH: Object File Path

Tools:	Compiler, Assembler, Linker, Decoder
Synonym:	None
Syntax:	"OBJPATH=" {<path>}.
Arguments:	<path>: Paths separated by semicolons, without spaces.
Default:	none.
Description	This environment variable is only relevant when object files are generated by the macro assembler. When this environment variable is defined, the assembler will store the object files it produces in the first directory specified there. If OBJPATH is not set, the generated object files will be stored in the directory the source file was found.
Example	OBJPATH=\sources\bin;..\..\headers;\usr\local\bin
See also	none.

SRECORD

SRECORD: S Record Type

Tools:	Assembler, Linker, Burner
Synonym:	None
Syntax:	"SRECORD=" <RecordType>.
Arguments:	<Record Type>: Force the type for the Motorola S record which must be generated. This parameter may take the value ‘S1’, ‘S2’ or ‘S3’.
Default:	none.
Description	This environment variable is only relevant when absolute files are directly generated by the macro assembler instead of object files. When this environment variable is defined, the assembler will generate a Motorola S file containing records from the specified type (S1 records when S1 is specified, S2 records when S2 is specified and S3 records when S3 is specified). *Note: If the environment variable SRECORD is set, it is the user responsibility to specify the appropriate S record type. If you specifies S1 while your code is loaded above 0xFFFF, the Motorola S file generated will not be correct, because the addresses will all be truncated to 2 bytes values.* When this variable is not set, the type of S record generated will depend on the size if the address, which must be loaded there. If the address can be coded on 2 bytes, a S1 record is generated. If the address is coded on 3 bytes, a S2 record is generated. Otherwise a S3 record is generated.
Example	SRECORD=S2
See also	none

TEXTPATH

TEXTPATH: Text File Path

Tools:	Compiler, Assembler, Linker, Decoder
Synonym:	none.
Syntax:	"TEXTPATH=" {<path>}.
Arguments:	<path>: Paths separated by semicolons, without spaces.
Default:	none.
Description	When this environment variable is defined, the assembler will store the listing files it produces in the first directory specified there. If TEXTPATH is not set, the generated listing files will be stored in the directory the source file was found.
Example	TEXTPATH=\sources\txt;..\..\headers;\usr\local\txt
See also	none.

TMP

TMP: Temporary directory

Tools:	Compiler, Assembler, Linker, Debugger, Librarian
Synonym:	none.
Syntax:	"TMP=" <directory>.
Arguments:	<directory>: Directory to be used for temporary files.
Default:	none.
Description	If a temporary file has to be created, normally the ANSI function tmpnam() is used. This library function stores the temporary files created in the directory specified by this environment variable. If the variable is empty or does not exist, the current directory is used. Check this variable if you get an error message “Cannot create temporary file”. *Note: This is an environment variable on system level (global environment variable) It CANNOT be specified in a default environment file (DEFAULT.ENV/.hidefaults).*
Example	TMP=C:\TEMP
See also	Section The Current Directory

USERNAME

USERNAME: User Name in Object File

Tools:	Compiler, Assembler, Linker, Librarian
Synonym:	none.
Syntax:	"USERNAME=" <user>.
Arguments:	<user>: Name of user.
Default:	none.
Description	Each object file contains an entry identifying the user who created the object file. This information may be retrieved from the object files using the decoder.
Example	USERNAME=PowerUser
See also	Environment variable COPYRIGHT
	Environment variable INCLUDETIME

Files

Input Files

Source Files

The macro assembler takes any file as input, it does not require the file name to have a special extension. However, we suggest that all your source file names have extension `.asm` and all included files extension `.inc`. Source files will be searched first in the project directory and then in the directories enumerated in `GENPATH`.

Include File

The search for include files is governed by the environment variable `GENPATH`. Include files are searched for first in the project directory, then in the directories given in the environment variable `GENPATH`. The project directory is set via the Shell, the Program Manager or the environment variable `DEFAULTDIR`.

Output Files

Object Files

After successful assembling session, the Macro Assembler generates an object file containing the target code as well as some debugging information. This file is written to the directory given in the environment variable `OBJPATH`. If that variable contains more than one path, the object file is written in the first directory given; if this variable is not set at all, the object file is written in the directory the source file was found. Object files always get the extension `.o`.

Absolute Files

When an application is encoded in a single module and all the sections are absolute sections, the user can decide to generate directly an absolute file instead of an object file. This file is written to the directory given in the environment variable `ABSPATH`. If that variable contains more than one path, the absolute file is written in the first directory given; if this variable is not set at all, the absolute file is written in the

directory the source file was found. Absolute files always get the extension `.abs`.

Motorola S Files

When an application is encoded in a single module and all the sections are absolute sections, the user can decide to generate directly an ELF absolute file instead of an object file. In that case a Motorola S record file is generated at the same time. This file can be burnt into an EPROM. It contains information stored in all the READ_ONLY sections in the application. The extension for the generated Motorola S record file depends on the setting from the variable SRECORD.

- If SRECORD = S1, the Motorola S record file gets the extension `.s1`.
- If SRECORD = S2, the Motorola S record file gets the extension `.s2`.
- If SRECORD = S3, the Motorola S record file gets the extension `.s3`.
- If SRECORD is not set, the Motorola S record file gets the extension `.sx`.

This file is written to the directory given in the environment variable ABSPATH. If that variable contains more than one path, the S record file is written in the first directory given; if this variable is not set at all, the S record file is written in the directory the source file was found.

Listing Files

After successful assembling session, the Macro Assembler generates a listing file containing each assembly instruction with their associated hexadecimal code. This file is always generated, when the option -L is activated (even when the macro assembler generates directly an absolute file). This file is written to the directory given in the environment variable TEXTPATH. If that variable contains more than one path, the listing file is written in the first directory given; if this variable is not set at all, the listing file is written in the directory the source file was found. Listing files always get the extension `.lst`. The format of the listing file is described in the Assembler Listing File chapter. This file is only generated when the option -L is activated.

Debug Listing Files

After successful assembling session, the Macro Assembler generates a debug listing file, which will be used to debug the application. This file is always generated, even when the macro assembler generates directly an absolute file. The debug listing file is a duplicate from the source, where all the macros are expanded and the include files merged. This file is written to the directory given in the environment variable

OBJPATH. If that variable contains more than one path, the debug listing file is written in the first directory given; if this variable is not set at all, the debug listing file is written in the directory the source file was found. Debug listing files always get the extension `.dbg`.

Error Listing File

If the Macro Assembler detects any errors, it does not create an object file but an error listing file. This file is generated in the directory the source file was found (also see *Environment*, *Environment Variable ERRORFILE*).

If the assembler's window is open, it displays the full path of all include files read. After successful assembling the number of code bytes generated is displayed, too. In case of error, the position and file name where the error occurs is displayed in the assembler window.

If the assembler is started from the *IDF* (with '`%f`' given on the command line) or Codewright (with '`%b%e`' given on the command line), this error file is not produced. Instead it writes the error messages in a special format in a file called `EDOUT` using the Microsoft format by default. Use *WinEdit*'s *Next Error* or CodeWright's *Find Next Error* command to see both error positions and the error messages.

Interactive Mode (Assembler window open)

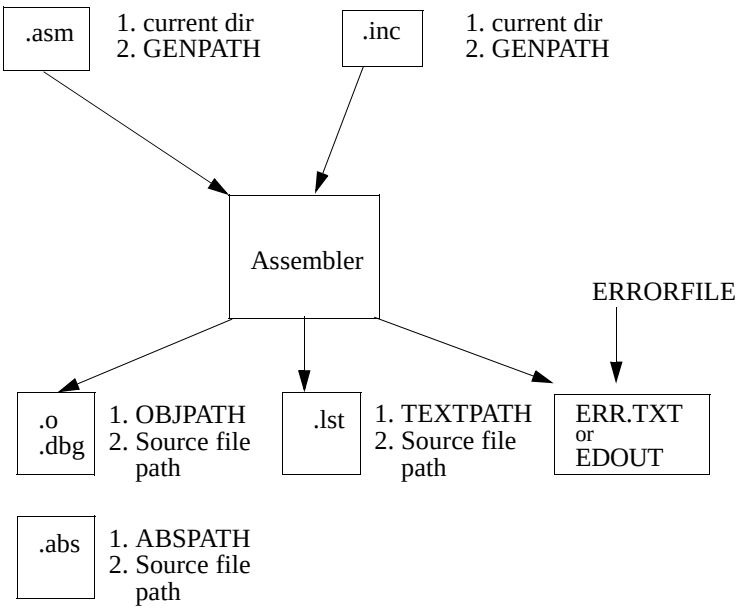
If `ERRORFILE` is set, the assembler creates a message file named as specified in this environment variable.

If `ERRORFILE` is not set, a default file named `ERR.TXT` is generated in the current directory.

Batch Mode (Assembler window not open)

If `ERRORFILE` is set, the assembler creates a message file named as specified in this environment variable.

If `ERRORFILE` is not set, a default file named `EDOUT` is generated in the current directory.


.asm
1. current dir
2. GENPATH
.inc
1. current dir
2. GENPATH
Assembler
ERRORFILE
.o
.dbg
1. OBJPATH
2. Source file path
.lst
1. TEXTPATH
2. Source file path
ERR.TXT
or
EDOUT
.abs
1. ABSPATH
2. Source file path

Assembler Options

The assembler offers a number of Assembler options that you can use to control the assembler's operation. Options are composed of a minus/dash ('-') followed by one or more letters or digits. Anything not starting with a dash/minus is supposed to be the name of a source file to be assembled. Assembler options may be specified on the command line or in the `ASMOPTIONS` environment variable. Typically, each Assembler option is specified only once per assembling session.

Command line options are not case sensitive, e.g. "`–Li`" is the same as "`–li`". It is possible to coalescing options in the same group, i.e. one might also write "`–Lci`" instead of "`–Lc –Li`". However such a usage is not recommended as it make the command line less readable and it does also create the danger of name conflicts. For example "`–Li –Lc`" is not the same as "`–Lic`" because this is recognized as a separate, independent option on its own.

Note: It is not possible to coalesce options in different groups, e.g. "`–Lc –W1`" cannot be abbreviated by the terms "`–LC1`" or "`–LCW1`".

`ASMOPTIONS` *If this environment variable is set, the assembler appends its contents to its command line each time a file is assembled. It can be used to globally specify certain options that should always be set, so you don't have to specify them each time a file is assembled.*

Assembler options are grouped by:
HOST, OUTPUT, INPUT, TARGET and VARIOUS.

Group	***Description***
HOST	*Lists options related to the host.*
OUTPUT	*Lists options related to the output files generation (which kind of file should be generated).*
INPUT	*Lists options related to the input files.*
CODE	*Lists options related to code generation (memory models, ...).*
MESSAGES	*Lists options controlling the generation of error messages.*
VARIOUS	*Lists various options.*

The group corresponds to the property sheets of the graphical option settings.

Each option has also a scope:

Scope	*Description*
Application	*The option has to be set for all files (Assembly Units) of an application. A typical example is an option to set the memory model. Mixing object files will have unpredictable results.*
Assembly Unit	*This option can be set for each assembling unit for an application differently. Mixing objects in an application is possible.*
None	*The option scope is not related to a specific code part. A typical example are options for the message management.*

The options available are arranged into different groups, and a sheet is available for each of these groups. The content of the list box depends on the selected sheets.

Assembler Option Details

The remainder of this section is devoted to describing each of the assembler options available for the assembler. The options are listed in alphabetical order and each is divided into several sections.

Topic	*Description*
Group	*HOST, OUTPUT, INPUT, CODE, LANGUAGE, MESSAGE or VARIOUS.*
Scope	*Application, Assembly Unit, Function or None.*
Syntax	*Specifies the syntax of the option in a EBNF format.*
Arguments	*Describes and lists optional and required arguments for the option.*
Default	*Shows the default setting for the option.*
Description	*Provides a detailed description of the option and how to use it.*

Topic	*Description*
Example	*Gives an example of usage, and effects of the option where possible. Assembler settings, source code and/or Linker PRM files are displayed where applicable. The examples shows an entry in the* `default.env` *for PC or in the* `.hidefaults` *for UNIX.*
See also	*Names related options.*

Using Special Modifiers

With some options, it is possible to use special modifiers. However, some modifiers may not make sense for all options. This section describes those modifiers.

The following modifiers are supported:

Modifier	**Description**
%p	Path including file separator
%N	File name in strict 8.3 format
%n	File name without extension
%E	Extension in strict 8.3 format
%e	Extension
%f	Path + file name without extension
%”	A double quote (“) if the file name, the path or the extension contains a space
%’	A single quote (‘) if the file name, the path or the extension contains a space
%(ENV)	Replaces it with the contents of an environment variable
%%	Generates a single ‘%’

Examples:

For your examples it is assumed that the actual file name (base file name for the modifiers) is:

```
c:\Metrowerks\my demo\TheWholeThing.myExt
```

%p gives the path only with a file separator:

```
c:\Metrowerks\my demo\
```

%N results in the file name in 8.3 format, that is the name with only 8 characters:

```
TheWhole
```

%n returns just the file name without extension:

```
TheWholeThing
```

%E gives the extension in 8.3 format, that is the extension with only 3 characters:

```
myE
```

%e is used for the whole extension:

```
myExt
```

%f gives the path plus the file name:

```
c:\Metrowerks\my demo\TheWholeThing
```

Because the path contains a space, using %” or %’ is recommended: Thus %”%f%” gives:

```
“c:\Metrowerks\my demo\TheWholeThing”
```

where %’%f%’ gives:

```
‘c:\Metrowerks\my demo\TheWholeThing’
```

Using %(envVariable) an environment variable may be used. A file separator following after % (envVariable) is ignored if the environment variable is empty or does not exist. In other words, the $(TEXTPATH)\myfile.txt is replaced with

```
c:\Metrowerks\txt\myfile.txt
```

if TEXTPATH is set to

```
TEXTPATH=c:\Metrowerks\txt
```

But is set to

```
myfile.txt
```

if TEXTPATH is does not exist or is empty.

A %% may be used to print a percent sign.The %e%% gives:

```
myExt%
```

List of all Options

-C=SAvocet	Semi Avocet Compatibility
-Ci	Switch Case Sensitivity on Label Names OFF
-Compat	Compatibility Modes
-CPU	CPU Derivative
-D	Define Label
-Env	Set Environment Variable
-F	Output File Format Specification
-H	Short Help
-I	Include File Path
-L	Generate a Listing File
-LasmC	Configure the Listing File
-Lc	No Macro Call in Listing File
-Ld	No Macro Definition in Listing File
-Le	No Macro Expansion in Listing File
-Li	No Included File in Listing File
-Lic	License Information
-LicA	License Information about every Feature in Directory
-MacroNest	Configure Maximum Macro Nesting
-M	Memory Model
-MCUasm	Switch Compatibility with MCUasm On
-N	Display Notify Box
-NoBeep	No Beep in Case of an Error
-NoDebugInfo	No debug info for ELF/Dwarf files
-NoEnv	Do not use Environment
-ObjN	Object File Name Specification
-Struct	Support for Structured Types
-V	Prints Assembler Version Number
-View	Application Standard Occurrence
-W1	No Information Messages
-W2	No Information and Warning Messages
-WErrFile	Create “err.log” Error File
-WMsg8x3	Cut File Names in Microsoft Format to 8.3
-WmsgFb	Set Message File Format for Batch Mode
-WmsgFi	Set Message File Format for Interactive Mode
-WmsgFob	Message Format for Batch Mode
-WmsgFoi	Message Format for Interactive Mode
-WmsgFonf	Message Format for no File Information
-WmsgFonp	Message Format for no Position Information

-WmsgNe	Number of Error Messages
-WmsgNi	Number of Information Messages
-WmsgNu	Disable User Messages
-WmsgNw	Number of Warning Messages
-WmsgSd	Setting a Message to Disable
-WmsgSe	Setting a Message to Error
-WmsgSi	Setting a Message to Information
-WmsgSw	Setting a Message to Warning
-WOutFile	Create Error Listing File
-WStdout	Write to Standard Output

-C=SAvocet

-C=SAvocet: Switch Semi-Compatibility with Avocet Assembler ON

Group:	VARIOUS
Scope:	Assembly Unit
Syntax:	"-C=SAvocet".
Arguments:	none
Default:	none.
Description:	This switches ON compatibility mode with the Avocet Assembler. Additional features supported, when this option is activated are enumerated in section “Semi-Avocet Compatibility”.
Example:	`ASMOPTIONS=-C=SAvocet`
See also:	Chapter “Semi-Avocet Compatibility”.

-Ci

-Ci: Switch Case Sensitivity on Label Names OFF

Group:	INPUT
Scope:	Assembly Unit
Syntax:	"-Ci".
Arguments:	none
Default:	none.
Description	This switches case sensitivity on label names off. When this option is activated, the assembler do not care about case sensitivity for label name. If the assembler generates object files and not directly absolute files (Option -FA2), then the case of exported/imported labels must still match. Or the option -Ci should be specified in the linker as well.

Example	When case sensitivity on label names is switched off, the assembler will not generate any error message for following code:
	
	`ORG $200`
	`entry: NOP`
	`BRA Entry`
	
	The instruction ‘BRA Entry’ will branch on the label ‘entry’.
	Per default, the assembler is case sensitive. For the assembler the labels ‘Entry’ and ‘entry’ are two distinct labels.

See also	Option -F

-CMacAngBrack

-CMacAngBrack: Angle brackets for Macro Arguments Grouping

Group:	LANGUAGE
Scope:	Application
Syntax:	"-CMacAngBrack" ("ON" | "OFF").
Arguments:	"ON" or "OFF".
Default:	none.
Description:	This option control whether the < > syntax for macro invocation argument grouping is available. When it is disabled, the assembler does not recognize the special meaning for < in the macro invocation context. There are cases where the angle brackets are ambiguous. New code should use the [? ?] syntax instead.
See also:	*Macro argument grouping* *Macros chapter* *Option -CMacBrackets*

-CMacBrackets

-CMacBrackets: Square brackets for Macro Arguments Grouping

Group:	LANGUAGE
Scope:	Application
Syntax:	"-CMacBrackets" ("ON" | "OFF").
Arguments:	"ON" or "OFF".
Default:	"ON".
Description:	This option control whether the [? ?] syntax for macro invocation argument grouping is available. When it is disabled, the assembler does not recognize the special meaning for [? in the macro invocation context.
See also:	*Macro argument grouping*
	Macros chapter
	Option -CMacAngBrack

-Compat

-Compat: Compatibility Modes

Group: LANGUAGE

Scope: Application

Syntax: "-Compat" ["=" {"!" | "=" | "c" | "s" | "f" | "$" | "a" | "b"}.

Arguments: see below.

Default: none.

Description: This option control some compatibility enhancements of the assembler. The goal is not to provide 100% compatibility with any other assembler, but to make it possible to reuse as much as possible. The various suboptions do control different parts of the assembly:

"=": Operator != means equal
The assembler takes the != operator by default as **not** equal, as it is in the C language. For compatibility, this behavior can be changed to equal with this option. Because the danger of this option for existing code, a message is issued for every != which is treated as equal.

"!": Support additional ! operators
The following additional operators are defined when this option is present:
"!^": exponentiation
"!m": modulo
"!@": signed greater or equal
"!g": signed greater
"!%": signed less or equal
"!t": signed less than
"!$": unsigned greater or equal
"!S": unsigned greater
"!&": unsigned less or equal
"!l": unsigned less
"!n": one complement
"!w": low operator
"!h": high operator

Note that the following ! operators are defined by default:
"!.": binary and
"!x": Exclusive or
"!+": binary or

"c": Alternate comment rules
With this suboption, comments do implicitly start when a space is present after the argument list. A special character is not necessary. Be careful with spaces when this option is given as part of the intended arguments may be taken as comment. However, to avoid accidental comments, the assembler does issue a warning if such a comment does not start with a "*" or a ";".

Example: Comments starting with a *

```
NOP   * With -Compat=c, comments
      * can start with a *
```

Example: Implicit comment start after a space
With -Compat=c, "+ 1" is taken as comment. A warning is issued because the "comment" does not start with a ";" or a "*".

```
DC.B 1 + 1 , 1
DC.B 1+1,1
```

With -Compat=C, this code generates a warning and the 3 bytes 1,2,1. Without it, this code generates the 4 bytes 2,1,2,1.

"s": Symbol prefixes
With this suboption, so compatibility prefixes for symbols are supported. With this option, the assembler does accept “pgz:” and “byte:” prefixed for symbols in XDEF’s and XREF’s. They correspond to a XREF.B or XDEF.B with the same symbol without the prefix.

"f": Ignore FF character at line start
With this suboption, an otherwise a illegal character recognized from feed character is ignored.

"$": Support $ character in symbols
With this suboption, the assembler supports to start identifiers with a $ sign.

"a": Add some additional directives
With this suboption, some additional directives are added for an enhanced compatibility.

The assembler does actually support a SECT directive as alias of the usual SECTION directive. The SECT directive takes the section name as first argument.

"b": support FOR directive
With this suboption, the assembler does support a FOR directive to generate repeated patterns more easily without having to use recursive macros.

See also: FOR directive

-CPUHC12, -CPUStar12

-CPU: Derivative

Group: CODE

Scope: Application

Syntax: "-CPU" {“HC12”|”Star12”}.

Arguments: none

Default: none.

Description: This option controls whether code for a HC12 or for a Star12 should be produced. Because the instruction set of the two CPUs is very similar, this option does only affect PCR relative MOVB/MOVW instructions. In HC12 or default mode, the assembler does adapt the offsets according to the CPU12 Reference Manual, paragraph 3.9.1 Move Instructions. In Star12 mode it does not.

Example: Consider the following code:

```
One:     DC 1
CopyOne: MOVB One, PCR, $1000
```

By default, or with -CPUHC12, the assembler generates:

```
000000 01          One:     DC 1
000001 180D DC10   CopyOne: MOVB One, PCR, $1000
003005 00
```

With the option -CPUStar12, the assembler generates:

```
003000 01          One:     DC 1
003001 180D DA10   CopyOne: MOVB One, PCR, $1000
003005 00
```

The difference is that for the HC12, the assembler adapts the offset to One according to the MOVB IDX/EXT case by -2, so the resulting code is $DC for the IDX encoding. For Star12, this is not done, and the IDX encodes as $DA.

Note: PC relative MOVB/MOVW instructions (e.g. “MOVB 1,PC,2,PC”) are not adapted. Only PCR relative move instructions (MOVB 1,PCR,2,PCR) are adapted.

See also: CPU12 Reference Manual, paragraph 3.9.1 Move Instructions

-D

-D: Define Label

Group: INPUT

Scope: Assembly Unit

Syntax: "-D" <LabelName> ["=" <Value>].

Arguments: <LabelName>: Name of label.
<Value>: Value for label. 0 if not present.

Default: 0 for the Value.

Description This option behaves as if a “Label: EQU Value” would be at the start of the main source file. When no explicit value is given, 0 is used as default.

This option can be used to build different versions with one common source file.

Example Conditional inclusion of a copyright notice:

Source file:

```
YearAsString:  MACRO
   DC.B $30+(\1 /1000)%10
   DC.B $30+(\1 / 100)%10
   DC.B $30+(\1 /  10)%10
   DC.B $30+(\1 /   1)%10
 ENDM

 ifdef ADD_COPYRIGHT
   ORG $1000
   DC.B "Copyright by "
   DC.B "John Doe"
 ifdef YEAR
   DC.B " 1999-"
   YearAsString YEAR
 endif
   DC.B 0
 endif
```

When assembled with the options "-dADD_COPYRIGHT -dYEAR=2001", the following listing is generated:

```
1   1                       YearAsString:  MACRO
```

```
 2   2                          DC.B $30+(\1 /1000)%10
 3   3                          DC.B $30+(\1 / 100)%10
 4   4                          DC.B $30+(\1 /  10)%10
 5   5                          DC.B $30+(\1 /   1)%10
 6   6                        ENDM
 7   7
 8   8          0000 0001     ifdef ADD_COPYRIGHT
 9   9                          ORG $1000
10  10  a001000 436F 7079       DC.B "Copyright by "
          001004 7269 6768
          001008 7420 6279
          00100C 20
11  11  a00100D 4A6F 686E       DC.B "John Doe"
          001011 2044 6F65
12  12          0000 0001     ifdef YEAR
13  13  a001015 2031 3939       DC.B " 1999-"
          001019 392D
14  14                           YearAsString YEAR
15   2m a00101B 32         +     DC.B $30+(YEAR /1000)%10
16   3m a00101C 30         +     DC.B $30+(YEAR / 100)%10
17   4m a00101D 30         +     DC.B $30+(YEAR /  10)%10
18   5m a00101E 31         +     DC.B $30+(YEAR /   1)%10
19  15                        endif
20  16  a00101F 00              DC.B 0
21  17                        endif
```

See also none.

-Env

-Env: Set Environment Variable

Group:	HOST
Scope:	Assembly Unit
Syntax:	"-Env" <EnvironmentVariable> "=" <VariableSetting>.
Arguments:	<EnvironmentVariable>: Environment variable to be set <VariableSetting>: Setting of the environment variable
Default:	none.
Description	This option sets an environment variable.

Example	`ASMOPTIONS=-EnvOBJPATH=\sources\obj`
	This is the same as
	`OBJPATH=\sources\obj`
	in the default.env.

See also	Environment Variable Details

-F (-Fh, -F2o, -FA2o, -F2, -FA2)

-F: Output File Format

Group:	OUTPUT
Scope:	Application
Syntax:	"-F" ("h" | "2o"| "A2o" | "2"| "A2").
Arguments:	"h": HIWARE object file format, this is the default
	"2o": Compatible ELF/DWARF 2.0 object file format
	"A2o": Compatible ELF/DWARF 2.0 absolute file format
	"2": ELF/DWARF 2.0 object file format
	"A2": ELF/DWARF 2.0 absolute file format
Default:	-F2
Description:	Define the format for the output file generated by the assembler.
	With the option -Fh set, the assembler uses an object file format which is proprietary of HIWARE.
	With the options -F2 set, the assembler produces an ELF/DWARF object file. This object file formats may also be supported by other Compiler or Assembler vendors.
	With the options -FA2 set, the assembler produces an ELF/DWARF absolute file. This file formats may also be supported by other Compiler or Assembler vendors.
	Note that the ELF/DWARF 2.0 files format has been updated in the current version of the assembler. If you are using HI-WAVE version 5.2 (or an earliest version), -F2o or -FA2o must be used to generate ELF/DWARF 2.0 object files which can be loaded in the debugger.
Example:	ASMOPTIONS=-F2
See also:	none.

-H

-H: Short Help

Group:	VARIOUS
Scope:	None
Syntax:	"-H".
Arguments:	none.
Default:	none.
Description	The -H option causes the assembler to display a short list (i.e. help list) of available options within the assembler window. Options are grouped into HOST, OUTPUT, INPUT, MESSAGE, CODE and VARIOUS. No other option or source files should be specified when the -H option is invoked.
Example	You find below a portion of the list produced by the option `-H`:

```
...
MESSAGE:
-N        Show notification box in case of errors
-NoBeep   No beep in case of an error
-W1       Don't print INFORMATION messages
-W2       Don't print INFORMATION or WARNING messages
-WErrFile Create "err.log" Error File
...
```

See also	none.

-I

-I: Include File Path

Group:	INPUT
Scope:	None
Syntax:	"-I"<path>.
Arguments:	<path>: File path to be used for includes.
Default:	none.
Description	With the option -I it is possible to specify a file path used for include files.
Example	-Id:\mySources\include
See also	none.

-L

-L: Generate a Listing File

Group: OUTPUT

Scope: Assembly unit

Syntax: "-L" ["=" <dest>]

Arguments: <dest>: the name of the listing file to be generated.
It may contain special modifiers (see Using Special Modifiers).

Default: no listing file generated.

Description Switches on the generation of the listing file. If dest is not specified, the listing file will have the same name as the source file, but with extension `.lst`. The listing file contains macro definition, invocation and expansion lines as well as expanded include files.

Example: `ASMOPTIONS=-L`

In the following example of assembly code, the macro cpChar accept two parameters. The macro copies the value of the first parameter to the second one.

When option -L is specified, following portion of code

```
         XDEF Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

With the following include file macro.inc

```
cpChar:  MACRO
         LDAA \1
         STAA \2
         ENDM
```

generates the following output in the assembly listing file:

```
 Abs. Rel.   Loc    Obj. code  Source line
```

```
---- ----   ------ ---------  -----------
   1    1                            XDEF Start
   2    2                     MyData:  SECTION
   3    3   000000            char1:   DS.B  1
   4    4   000001            char2:   DS.B  1
   5    5                           INCLUDE "macro.inc"
   6    1i                    cpChar:  MACRO
   7    2i                             LDAA \1
   8    3i                             STAA \2
   9    4i                             ENDM
  10    6                     CodeSec: SECTION
  11    7                     Start:
  12    8                           cpChar char1, char2
  13    2m  000000 B6 xxxx   +         LDAA char1
  14    3m  000003 7A xxxx   +         STAA char2
  15    9   000006 A7                   NOP
```

Content of included files, as well as macro definition, invocation and expansion is stored in the listing file.

For a detailed description of the listing file, see the Listing File chapter.

See also

Option -Lasmc
Option -Lc
Option -Ld
Option -Le
Option -Li

-Lasmc

-Lasmc: Configure Listing File

Group: OUTPUT

Scope: Assembly unit

Syntax: "-Lasmc" "=" [{"s" | "r" | "m" | "l" | "k" | "i" | "c" | "a"}.

Arguments:

s: Do not write the source line
r: Do not write the relative line (Rel.)
m: Do not write the macro mark
l: Do not write the address (Loc)
k: Do not write the location kind
i: Do not write the include mark column
c: Do not write the object code
a: Do not write the absolute line (Abs.)

Default: Write all columns.

Description The listing file shows by default a lot of information. With this option, the output can be reduced to columns which are of interrest. This option configures which columns are printed in a listing file. To configure which lines to print, see the options Option -Lc Option -Ld, Option -Le and Option -Li.

Example: For the following file:

```
DC.B "Hello World"
DC.B 0
```

The assembler generates by default this listing file:

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1   000000 4865 6C6C       DC.B "Hello World"
           000004 6F20 576F
           000008 726C 64
   2    2   00000B 00              DC.B 0
```

In order to get this output without the source file line numbers and other irrelevant parts for this simple DC.B example, the following option is added "`-Lasmc=ramki`", this generates:

```
Loc    Obj. code Source line
------ --------- -----------
000000 4865 6C6C     DC.B "Hello World"
000004 6F20 576F
000008 726C 64
00000B 00            DC.B 0
```

For a detailed description of the listing file, see the Listing File chapter.

See also

Option -L
Option -Lc
Option -Ld
Option -Le
Option -Li
Listing File chapter

-Lc

-Lc: No Macro Call in Listing File

Group: OUTPUT

Scope: Assembly unit

Syntax: "-Lc"

Arguments: none.

Default: none.

Description Switches on the generation of the listing file, but macro invocations are not present in the listing file. The listing file contains macro definition and expansion lines as well as expanded include files.

Example: `ASMOPTIONS=-Lc`

In the following example of assembly code, the macro cpChar accept two parameters. The macro copies the value of the first parameter to the second one.

When option -Lc is specified, following portion of code

```
         XDEF Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

With the include file macro.inc:

```
cpChar:  MACRO
         LDAA \1
         STAA \2
         ENDM
```

generates the following output in the assembly listing file:

```
 Abs. Rel.   Loc    Obj. code  Source line
 ---- ----   ------ ---------  -----------
    1    1                             XDEF Start
```

```
 2    2                      MyData:  SECTION
 3    3   000000             char1:   DS.B  1
 4    4   000001             char2:   DS.B  1
 5    5                             INCLUDE "macro.inc"
 6    1i                     cpChar:  MACRO
 7    2i                              LDAA \1
 8    3i                              STAA \2
 9    4i                              ENDM
10    6                      CodeSec: SECTION
11    7                      Start:
13    2m  000000 B6 xxxx   +          LDAA char1
14    3m  000003 7A xxxx   +          STAA char2
15    9   000006 A7                  NOP
```

Content of included files, as well as macro definition and expansion is stored in the listing file.

The source line containing the invocation of the macro is not present in the listing file.

For a detailed description of the listing file, see the Listing File chapter.

See also

Option -L
Option -Ld
Option -Le
Option -Li

-Ld

-Ld: No Macro Definition in Listing File

Group: OUTPUT

Scope: Assembly unit

Syntax: "-Ld"

Arguments: none.

Default: none.

Description Switches on the generation of the listing file, but macro definitions are not present in the listing file. The listing file contains macro invocation and expansion lines as well as expanded include files.

Example: `ASMOPTIONS=-Ld`

In the following example of assembly code, the macro cpChar accept two parameters. The macro copies the value of the first parameter to the second one.

When option -Ld is specified, following portion of code

```
         XDEF Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

With the include file macro.inc:

```
cpChar:  MACRO
         LDAA \1
         STAA \2
         ENDM
```

generates the following output in the assembly listing file:

```
 Abs. Rel.   Loc    Obj. code  Source line
 ---- ----   ------ ---------  -----------
```

```
 1    1                                  XDEF Start
 2    2                        MyData:   SECTION
 3    3   000000               char1:    DS.B  1
 4    4   000001               char2:    DS.B  1
 5   5                                  INCLUDE "macro.inc"
 6    1i                       cpChar:   MACRO
10    6                        CodeSec:  SECTION
11    7                        Start:
12   8                                   cpChar char1, char2
13    2m  000000 B6 xxxx   +              LDAA char1
14    3m  000003 7A xxxx   +              STAA char2
15    9   000006 A7                      NOP
```

Content of included files, as well as macro invocation and expansion is stored in the listing file.

The source code from the macro definition is not present in the listing file.

For a detailed description of the listing file, see the Listing File chapter.

See also

Option -L
Option -Lc
Option -Le
Option -Li

-Le

-Le: No Macro Expansion in Listing File

Group: OUTPUT

Scope: Assembly unit

Syntax: "-Le"

Arguments: none.

Default: none.

Description Switches on the generation of the listing file, but macro expansions are not present in the listing file. The listing file contains macro definition and invocation lines as well as expanded include files.

Example: `ASMOPTIONS=-Le`

In the following example of assembly code, the macro cpChar accept two parameters. The macro copies the value of the first parameter to the second one.

When option -Le is specified, following portion of code

```
         XDEF Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

With the include file macro.inc:

```
cpChar:  MACRO
         LDAA \1
         STAA \2
         ENDM
```

generates the following output in the assembly listing file:

```
 Abs. Rel.   Loc    Obj. code  Source line
 ---- ----   ------ ---------  -----------
```

```
 1    1                                 XDEF Start
 2    2                          MyData: SECTION
 3    3   000000                 char1:  DS.B  1
 4    4   000001                 char2:  DS.B  1
 5    5                                INCLUDE "macro.inc"
 6    1i                         cpChar: MACRO
 7    2i                                 LDAA \1
 8    3i                                 STAA \2
 9    4i                                 ENDM
10    6                          CodeSec: SECTION
11    7                          Start:
12    8                                cpChar char1, char2
15    9   000006 A7                      NOP
```

Content of included files, as well as macro definition and invocation are stored in the listing file.

The macro expansion lines are not present in the listing file.

For a detailed description of the listing file, see the Listing File chapter.

See also

Option -L
Option -Lc
Option -Ld
Option -Li

-Li

-Li: No included File in Listing File

Group: OUTPUT

Scope: Assembly unit

Syntax: "-Li"

Arguments: none.

Default: none.

Description Switches on the generation of the listing file, but include files are not expanded in the listing file. The listing file contains macro definition, invocation and expansion lines.

Example: `ASMOPTIONS=-Li`

In the following example of assembly code, the macro cpChar accept two parameters. The macro copies the value of the first parameter to the second one.

When option -Li is specified, following portion of code

```
         XDEF Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

With the include file macro.inc:

```
cpChar:  MACRO
         LDAA \1
         STAA \2
         ENDM
```

generates the following output in the assembly listing file:

```
 Abs. Rel.   Loc    Obj. code  Source line
 ---- ----   ------ ---------  -----------
    1    1                           XDEF Start
    2    2                    MyData:  SECTION
```

```
 3    3   000000            char1:   DS.B  1
 4    4   000001            char2:   DS.B  1
 5    5                           INCLUDE "macro.inc"
10    6                     CodeSec: SECTION
11    7                     Start:
12    8                           cpChar char1, char2
13    2m  000000 B6 xxxx    +        LDAA char1
14    3m  000003 7A xxxx    +        STAA  char2
15    9   000006 A7                  NOP
```

Macro definition, invocation and expansion is stored in the listing file.

The content of included file is not present in the listing file.

For a detailed description of the listing file, see the Listing File chapter.

See also

Option -L
Option -Lc
Option -Ld
Option -Le

-Lic

-Lic: License Information

Group:	VARIOUS
Scope:	None
Syntax:	"-Lic".
Arguments:	none.
Default:	none.
Description	The -Lic option prints the current license information (e.g. if it is a demo version or a full version). This information is also displayed in the about box.
Example	`ASMOPTIONS=-Lic`
See also	Option -LicA

-LicA

-LicA: License Information about every Feature in Directory

Group:	VARIOUS
Scope:	None
Syntax:	"-LicA".
Arguments:	none.
Default:	none.
Description	The -LicA option prints the license information of every tool or DLL in the directory where the executable is (e.g. if tool or feature is a demo version or a full version). Because the option has to analyze every single file in the directory, this may take a long time.
Example	`ASMOPTIONS=-LicA`
See also	Option -Lic

-M (-Ms, -Mb, -Ml)

-M: Memory Model

Group:	CODE
Scope:	Application
Syntax:	"-M" ("s" | "b" | "l").
Arguments:	"s": small memory model "b": banked memory model "l": large memory model.
Default:	-Ms
Description:	The assembler for the MC68HC12 supports three different memory models. Default is the small memory model, which corresponds to the normal setup, i.e. a 64kB code-address space. If you use some code memory expansion scheme, you may use banded memory model. The large memory model is used when using both code and data memory expansion scheme. Memory models are interesting when mixing ANSI-C and assembler files. For compatibility reasons, the memory model used by the different files must be identical.
Example:	`ASMOPTIONS=-Ms`
See also:	none.

-MacroNest

-MacroNest: Configure Maximum Macro Nesting

Group:	Language
Scope:	Assembly Unit
Syntax:	"-MacroNest" <Value>.
Arguments:	<Value>: max. allowed nesting level.
Default:	3000.
Description	This option controls how deep macros calls can be nested. It's main purpose is to avoid endless recursive macro invocations. When the nesting level is reached, then the message A
Example	See the description of message A1004 for an example.
See also	Message A1004

-MCUasm

-MCUasm: Switch Compatibility with MCUasm ON

Group:	VARIOUS
Scope:	Assembly Unit
Syntax:	"-MCUasm".
Arguments:	none
Default:	none.
Description	This switches ON compatibility mode with the MCUasm Assembler. Additional features supported, when this option is activated are enumerated in section “MCUasm Compatibility”.
Example	`ASMOPTIONS=-MCUasm`
See also	Chapter MCUasm Compatibility.

-N

-N: Display Notify Box

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-N".
Arguments:	none.
Default:	none.
Description	Makes the assembler display an alert box if there was an error during assembling. This is useful when running a makefile (please see Manual about *Make Utility*) since the assembler waits for the user to acknowledge the message, thus suspending makefile processing. (The 'N' stands for “Notify”.) This feature is useful for halting and aborting a build using the Make Utility.
Example	`ASMOPTIONS=-N` If during assembling an error occurs, a dialog box will be opened.
See also	none.

-NoBeep

-NoBeep: No Beep in Case of an Error

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-NoBeep".
Arguments:	none.
Default:	none.
Description	Normally there is a ‘beep’ notification at the end of processing if there was an error. To have a silent error behavior, this ‘beep’ may be switched off using this option.
Example	`ASMOPTIONS=-NoBeep`
See also	none.

-NoDebugInfo

-NoDebugInfo: No Debug Information for ELF/ Dwarf Files

Group:	LANGUAGE
Scope:	Assembly Unit
Syntax:	"-NoDebugInfo".
Arguments:	none.
Default:	none.
Description	By default, the assembler produces debugging info for the produced ELF/Dwarf files. With this option this can be switched off.
Example	`ASMOPTIONS=-NoDebugInfo`
See also	none.

-NoEnv

-NoEnv: Do not use Environment

Group:	Startup. (This option cannot be specified interactively)
Scope:	Assembly Unit
Syntax:	"-NoEnv".
Arguments:	none.
Default:	none.
Description	This option can only be specified at the command line while starting the application. It cannot be specified in any other circumstances, including the default.env file, the command line or whatever. When this option is given, the application does not use any environment (default.env, project.ini or tips file).
Example	`xx.exe -NoEnv` (use the actual executable name instead of “xx”)
See also	Environment

-ObjN

-ObjN: Object File Name Specification

Group:	OUTPUT
Scope:	Assembly Unit
Syntax:	"-ObjN"<FileName>.
Arguments:	<FileName>: Name of the binary output file generated.
Default:	-ObjN%n.o (when relocatable file generated),| -ObjN%n.abs (when absolute file generated).
Description	Normally, the object file has the same name than the processed source file, but with extension ".o" when relocatable code is generated or ".abs" when absolute code is generated. This option allows a flexible way to define the output file name. The modifier "%n" can be used, it is replaced with the source file name. If <file> in the option contains a path (absolute or relative), the environment variable OBJPATH is ignored.

Example	`ASMOPTIONS=-ObjNa.out`
	The resulting object file will be "a.out". If the environment variable OBJPATH is set to "\src\obj", the object file will be"\src\obj\a.out".
	`fibo.c -ObjN%n.obj`
	The resulting object file will be "fibo.obj".
	`myfile.c -ObjN..\objects\_%n.obj`
	The object file will be named relative to the current directory to "..\objects_myfile.obj. Note that the environment variable OBJPATH is ignored, because the <file> contains a path.

See also	Environment variable OBJPATH.

-Prod

-Prod: Specify Project File at Startup

Group:	none. (This option cannot be specified interactively)
Scope:	none.
Syntax:	"-Prod=” <file>.
Arguments:	<file>: name of a project or project directory
Default:	none.
Description	This option can only be specified at the command line while starting the application. It cannot be specified in any other circumstances, including the default.env file, the command line or whatever. When this option is given, the application opens the file as configuration file. When the file name does only contain a directory, the default name project.ini is appended. When the loading fails, a message box appears.
Example	`assembler.exe -prod=project.ini` (use the assembler executable name instead of “assembler”)
See also	Environment

-Struct

-Struct: Support for Structured Types

Group:	INPUT
Scope:	Assembly Unit
Syntax:	"-Struct".
Arguments:	None.
Default:	None
Description	When this option is activated, the macro assembler also support the definition and usage of structured types. This is interesting for application containing both ANSI C and Assembly modules.
Example	`ASMOPTIONS=-Struct`
See also	Chapter Mix C and Assembler Applications.

-V

-V: Prints the Assembler Version

Group:	VARIOUS
Scope:	None
Syntax:	"-V".
Arguments:	none.
Default:	none.
Description	Prints the assembler version and the current directory
	Note: This option is useful to determine the current directory of the assembler
Example	-V produces the following list:

```
Command Line '-v'
Assembler V-5.0.8, Jul  7 1998

Directory: C:\metrowerks\demo

Common Module V-5.0.7, Date Jul  7 1998
User Interface Module, V-5.0.17, Date Jul  7 1998
Assembler Kernel, V-5.0.13, Date Jul  7 1998
Assembler Target, V-5.0.8, Date Jul  7 1998
```

See also	none.

-View

-View: Application Standard Occurrence

Group:	HOST
Scope:	Assembly Unit
Syntax:	"-View" <kind>.
Arguments:	<kind> is one of: “Window”: Application window has default window size “Min”: Application window is minimized “Max”: Application window is maximized “Hidden”: Application window is not visible (only if arguments)
Default:	Application started with arguments: Minimized. Application started without arguments: Window.
Description	Normally the application (e.g. assembler, linker, compiler, ...) is started as normal window if no arguments are given. If the application is started with arguments (e.g. from the maker to assemble/link/compile a file) then the application is running minimized to allow batch processing. However, with this option the behavior may be specified. Using -ViewWindow the application is visible with its normal window. Using -ViewMin the application is visible iconified (in the task bar). Using -ViewMax the application is visible maximized (filling the hole screen). Using -ViewHidden the application processes arguments (e.g. files to be compiled/linked) completely invisible in the back ground (no window/icon in the task bar visible). However e.g. if you are using the -N option a dialog box is still possible.
Example	c:\metrowerks\prog\linker.exe -ViewHidden fibo.prm
See also	none.

-W1

-W1: No Information Messages

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-W1".
Arguments:	none.
Default:	none.
Description	Inhibits the assembler’s printing INFORMATION messages, only WARNING and ERROR messages are written to the error listing file and to the assembler window.
Example	`ASMOPTIONS=-W1`
See also	none.

-W2

-W2: No Information and Warning Messages

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-W2".
Arguments:	none.
Default:	none.
Description	Suppresses all messages of type INFORMATION and WARNING, only ERRORs are written to the error listing file and to the assembler window .
Example	`ASMOPTIONS=-W2`
See also	none.

-WErrFile

-WErrFile: Create "err.log" Error File

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WErrFile" ("On" | "Off").
Arguments:	none.
Default:	err.log is created/deleted.
Description	The error feedback from the assembler to called tools is now done with a return code. In 16 bit windows environments, this was not possible, so in the error case a file “err.log” with the numbers of errors written into was used to signal an error. To state no error, the file “err.log” was deleted. Using UNIX or WIN32, there is now a return code available, so this file is no longer needed when only UNIX / WIN32 applications are involved. To use a 16 bit maker with this tool, the error file must be created in order to signal any error.

Example	`-WErrFileOn`
	err.log is created/deleted when the application is finished.
	`-WErrFileOff`
	existing err.log is not modified.

See also	Option -WStdout Option -WOutFile

-Wmsg8x3

-Wmsg8x3: Cut File Names in Microsoft Format to 8.3

Group: MESSAGE

Scope: Assembly Unit

Syntax: "-Wmsg8x3".

Arguments: none.

Default: none.

Description: Some editors (e.g. early versions of WinEdit) are expecting the file name in the Microsoft message format in a strict 8.3 format, that means the file name can have at most 8 characters with not more than a 3 characters extension. Using Win95 or WinNT longer file names are possible. With this option the file name in the Microsoft message is truncated to the 8.3 format.

Example

```
x:\mysourcefile.c(3): INFORMATION C2901: Unrolling
loop
```

With the option -Wmsg8x3 set, the above message will be

```
x:\mysource.c(3): INFORMATION C2901: Unrolling loop
```

See also

Option -WmsgFi
Option -WmsgFb
Option -WmsgFoi
Option -WMsgFob
Option -WmsgFonP

-WmsgCE

-WmsgCE: RGB color for error messages

Group:	MESSAGE
Scope:	Compilation Unit
Syntax:	"-WmsgCE" <RGB>.
Arguments:	<RGB>: 24bit RGB (red green blue) value.
Default:	-WmsgCE16711680 (rFF g00 b00, red)
Description	With this options it is possible to change the error message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.
Example	-WmsgCE255 changes the error messages to blue.
See also	none.

-WmsgCF

-WmsgCF: RGB color for fatal messages

Group:	MESSAGE
Scope:	Compilation Unit
Syntax:	"-WmsgCF" <RGB>.
Arguments:	<RGB>: 24bit RGB (red green blue) value.
Default:	-WmsgCF8388608 (r80 g00 b00, dark red)
Description	With this options it is possible to change the fatal message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.
Example	-WmsgCF255 changes the fatal messages to blue.
See also	none.

-WmsgCI

-WmsgCI: RGB color for information messages

Group:	MESSAGE
Scope:	Compilation Unit
Syntax:	"-WmsgCI" <RGB>.
Arguments:	<RGB>: 24bit RGB (red green blue) value.
Default:	-WmsgCI32768 (r00 g80 b00, green)
Description	With this options it is possible to change the information message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.
Example	-WmsgCI255 changes the information messages to blue.
See also	none.

-WmsgCU

-WmsgCU: RGB color for user messages

Group:	MESSAGE
Scope:	Compilation Unit
Syntax:	"-WmsgCU" <RGB>.
Arguments:	<RGB>: 24bit RGB (red green blue) value.
Default:	-WmsgCU0 (r00 g00 b00, black)
Description	With this options it is possible to change the user message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.
Example	-WmsgCU255 changes the user messages to blue.
See also	none.

-WmsgCW

-WmsgCW: RGB color for warning messages

Group:	MESSAGE
Scope:	Compilation Unit
Syntax:	"-WmsgCW" <RGB>.
Arguments:	<RGB>: 24bit RGB (red green blue) value.
Default:	-WmsgCW255 (r00 g00 bFF, blue)
Description	With this options it is possible to change the warning message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.
Example	-WmsgCW0 changes the warning messages to black.
See also	none.

-WmsgFb (-WmsgFbv, -WmsgFbm)

-WmsgFb: Set Message File Format for Batch Mode

Group: MESSAGE

Scope: Assembly Unit

Syntax: "-WmsgFb" ["v" | "m"].

Arguments: "v": Verbose format.

"m": Microsoft format.

Default: -WmsgFbm

Description The assembler can be started with additional arguments (e.g. files to be assembled together with Assembler options). If the assembler has been started with arguments (e.g. from the *Make Tool* or with the '%f' argument from the IDF), the assembler assembles the files in a batch mode, that is no Assembler window is visible and the assembler terminates after job completion.

If the assembler is in batch mode the assembler messages are written to a file and not visible on the screen. This file only contains the assembler messages (see examples below).

By default, the assembler uses a *Microsoft* message format to write the assembler messages (errors, warnings, information messages) if the assembler is in batch mode.

With this option, the default format may be changed from the *Microsoft* format (only line information) to a more verbose error format with line, column and source information

Example

```
var1:     equ   5
var2:     equ   5
   if (var1=var2)
         NOP
   endif
   endif
```

By default, the assembler generates the following error output in the assembler window if it is running in batch mode:

```
X:\TW2.ASM(12):ERROR: Conditional else not allowed here
```

Setting the format to verbose, more information is stored in the file:

```
ASMOPTIONS=-WmsgFbv
>> in "C:\tw2.asm", line 6, col 0, pos 81
   endif
^
ERROR A1001: Conditional else not allowed here
```

See also

Environment variable ERRORFILE
Option -WmsgFob
Option -WmsgFi
Option -WmsgFonp
Option -WmsgFoi
Option -WmsgFonf

-WmsgFi (-WmsgFiv, -WmsgFim)

-WmsgFi: Set Message File Format for Interactive Mode

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WmsgFi" ["v" | "m"].
Arguments:	"v": Verbose format.
	"m": Microsoft format.
Default:	-WmsgFiv
Description	If the assembler is started without additional arguments (e.g. files to be assembled together with Assembler options), the assembler is in the interactive mode (that is, a window is visible).
	By default, the assembler uses the verbose error file format to write the assembler messages (errors, warnings, information messages).
	With this option, the default format may be changed from the verbose format (with source, line and column information) to the *Microsoft* format (only line information).
	With this option, the default format may be changed from the *Microsoft* format (only line information) to a more verbose error format with line, column and source information.
	Note: Using the Microsoft format may speed up the assembly process, because the assembler has to write less information to the screen.
Example	By default, the assembler following error output in the assembler window if it is running in interactive mode.

```
>> in "X:\TWE.ASM", line 12, col 0, pos 215
         endif
         endif
^
ERROR A1001: Conditional else not allowed here
```

Setting the format to Microsoft, less information is displayed:

```
ASMOPTIONS=-WmsgFim
X:\TWE.ASM(12): ERROR: conditional else not allowed here
```

See also

Environment variable ERRORFILE
Option -WmsgFob
Option -WmsgFb
Option -WmsgFonp
Option -WmsgFoi
Option -WmsgFonf

-WmsgFob

-WmsgFob: Message Format for Batch Mode

Group: MESSAGE

Scope: Assembly Unit

Syntax: "-WmsgFob"<string>.

Arguments: <string>: format string (see below).

Default: -WmsgFob"%f%e(%l): %K %d: %m\n"

Description: With this option it is possible modify the default message format in batch mode. The following formats are supported (supposed that the source file is `x:\metrowerks\sourcefile.asmx`)

```
Format Description        Example
-----------------------------------
%s     Source Extract
%p     Path                x:\metrowerks\
%f     Path and name       x:\metrowerks\sourcefile
%n     File name           sourcefile
%e     Extension           .asmx
%N     File (8 chars)      sourcefi
%E     Extension (3 chars) .asm
%l     Line                3
%c     Column              47
%o     Pos                 1234
%K     Uppercase kind      ERROR
%k     Lowercase kind      error
%d     Number              A1051
%m     Message             text
%%     Percent             %
\n     New line
```

Example: `ASMOPTIONS=-WmsgFob”%f%e(%l): %k %d: %m\n”`

produces a message in following format:

```
x:\metrowerks\sourcefile.asmx(3): error A1051: Right
parenthesis expected
```

See also: Environment variable ERRORFILE
Option -WmsgFb
Option -WmsgFi
Option -WmsgFonp

Option -WmsgFonf
Option -WmsgFoi

-WmsgFoi

-WmsgFoi: Message Format for Interactive Mode

Group: MESSAGE

Scope: Assembly Unit

Syntax: "-WmsgFoi"<string>.

Arguments: <string>: format string (see below).

Default: -WmsgFoi"\n>> in \"%f%e\", line %l, col %c, pos %o\n%s\n%K %d: %m\n"

Description: With this option it is possible modify the default message format in interactive mode. The following formats are supported (supposed that the source file is `x:\metrowerks\sourcefile.asmx`):

```
Format Description        Example
-------------------------------------
%s     Source Extract
%p     Path                x:\metrowerks\
%f     Path and name       x:\metrowerks\sourcefile
%n     File name           sourcefile
%e     Extension           .asmx
%N     File (8 chars)      sourcefi
%E     Extension (3 chars) .asm
%l     Line                3
%c     Column              47
%o     Pos                 1234
%K     Uppercase kind      ERROR
%k     Lowercase kind      error
%d     Number              A1051
%m     Message             text
%%     Percent             %
\n     New line
```

Example: `ASMOPTIONS=-WmsgFoi"%f%e(%l): %k %d: %m\n"`

produces a message in following format:

```
x:\metrowerks\sourcefile.asmx(3): error A1051: Right
parenthesis expected
```

See also
- Environment variable ERRORFILE
- Option -WmsgFb
- Option -WmsgFi
- Option -WmsgFonp
- Option -WmsgFonf
- Option -WmsgFob

-WmsgFonf

-WmsgFonf: Message Format for no File Information

Group: MESSAGE

Scope: Assembly Unit

Syntax: "-WmsgFonf"<string>.

Arguments: <string>: format string (see below).

Default: -WmsgFonf"%K %d: %m\n"

Description: Sometimes there is no file information available for a message (e.g. if a message not related to a specific file). Then this message format string is used. The following formats are supported:

```
Format Description        Example
-----------------------------------
%K     Uppercase kind     ERROR
%k     Lowercase kind     error
%d     Number             L10324
%m     Message            text
%%     Percent            %
\n     New line
```

Example: `ASMOPTIONS=-WmsgFonf”%k %d: %m\n”`

produces a message in following format:

```
information L10324: Linking successful
```

See also: Environment variable ERRORFILE
Option -WmsgFb
Option -WmsgFi
Option -WmsgFonp
Option -WmsgFoi
Option -WmsgFob

-WmsgFonp

-WmsgFonp: Message Format for no Position Information

Group: MESSAGE

Scope: Assembly Unit

Syntax: "-WmsgFonp"<string>.

Arguments: <string>: format string (see below).

Default: -WmsgFonp"%f%e: %K %d: %m\n"

Description: Sometimes there is no position information available for a message (e.g. if a message not related to a certain position). Then this message format string is used. The following formats are supported (supposed that the source file is x:\metrowerks\sourcefile.asmx)

```
Format Description        Example
-----------------------------------
%p     Path               x:\metrowerks\
%f     Path and name      x:\metrowerks\sourcefile
%n     File name          sourcefile
%e     Extension          .asmx
%N     File (8 chars)     sourcefi
%E     Extension (3 chars) .asm
%K     Uppercase kind     ERROR
%k     Lowercase kind     error
%d     Number             L10324
%m     Message            text
%%     Percent            %
\n     New line
```

Example: `ASMOPTIONS=-WmsgFonf"%k %d: %m\n"`

produces a message in following format:

```
information L10324: Linking successful
```

See also:
Environment variable ERRORFILE
Option -WmsgFb
Option -WmsgFi
Option -WmsgFonf
Option -WmsgFoi

Option -WmsgFonfob

-WmsgNe

-WmsgNe: Number of Error Messages

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WmsgNe" <number>.
Arguments:	<number>: Maximum number of error messages.
Default:	50
Description	With this option the amount of error messages can be reported until the assembler stops assembling. Note that subsequent error messages which depends on a previous one may be confusing.
Example	`ASMOPTIONS=-WmsgNe2` The assembler stops assembling after two error messages.
See also	Option -WmsgNi Option -WmsgNw

-WmsgNi

-WmsgNi: Number of Information Messages

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WmsgNi" <number>.
Arguments:	<number>: Maximum number of information messages.
Default:	50
Description	With this option the amount of information messages can be set.
Example	`ASMOPTIONS=-WmsgNi10` Only ten information messages are logged.
See also	Option -WmsgNe Option -WmsgNw

-WmsgNu

-WmsgNu: Disable User Messages

Group:	MESSAGE
Scope:	None.
Syntax:	"-WmsgNu" ["=" {"a" | "b" | "c" | "d"}].
Arguments:	“a”: Disable messages about include files “b”: Disable messages about reading files “c”: Disable messages about generated files “d”: Disable messages about processing statistics “e”: Disable informal messages
Default:	none.
Description	The application produces some messages which are not in the normal message categories (WARNING, INFORMATION, WRROR, FATAL). With this option such messages can be disabled. The idea of this option is to reduce the amount of messages and to simplify the error parsing of other tools. “a”: The application informs about all included files. With this suboption this can be disabled. “b”: With this suboption messages about reading files e.g. the files used as input can be disabled. “c”: Disables messages informing about generated files. “d”: At the end the application may inform about statistics, e.g. code size, RAM/ROM usage and so on. With this suboption this can be disabled. “e”: With this option informal messages (e.g. memory model, floating point format, ...) can be disabled. *Note: Depending on the application, not all suboptions may make sense. In this case they are just ignored for compatibility.*
Example	`-WmsgNu=c`
See also	none.

-WmsgNw

-WmsgNw: Number of Warning Messages

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WmsgNw" <number>.
Arguments:	<number>: Maximum number of warning messages.
Default:	50
Description	With this option the amount of warning messages can be set.
Example	`ASMOPTIONS=-WmsgNw15`
	Only 15 warning messages are logged.
See also	Option -WmsgNe Option -WmsgNi

-WmsgSd

-WmsgSd: Setting a Message to Disable

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WmsgSd" <number>.
Arguments:	<number>: Message number to be disabled, e.g. 1801
Default:	none.
Description	With this option a message can be disabled, so it does not appear in the error output.
Example	`-WmsgSd1801`
See also	Option -WmsgSi Option -WmsgSw Option -WmsgSe

-WmsgSe

-WmsgSe: Setting a Message to Error

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WmsgSe" <number>.
Arguments:	<number>: Message number to be an error, e.g. 1853
Default:	none.
Description	Allows changing a message to an error message.
Example	`-WmsgSe1853`
See also	Option -WmsgSd Option -WmsgSi Option -WmsgSw

-WmsgSi

-WmsgSi: Setting a Message to Information

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WmsgSi" <number>.
Arguments:	<number>: Message number to be an information, e.g. 1853
Default:	none.
Description	With this option a message can be set to an information message.
Example	`-WmsgSi1853`
See also	Option -WmsgSd Option -WmsgSw Option -WmsgSe

-WmsgSw

-WmsgSw: Setting a Message to Warning

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WmsgSw" <number>.
Arguments:	<number>: Error number to be a warning, e.g. 2901
Default:	none.
Description	With this option a message can be set to a warning message.
Example	`-WmsgSw2901`
See also	Option -WmsgSd Option -WmsgSi Option -WmsgSe

-WOutFile

-WOutFile: Create Error Listing File

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WOutFile" ("On" | "Off").
Arguments:	none.
Default:	Error listing file is created.
Description	This option controls if a error listing file should be created at all. The error listing file contains a list of all messages and errors which are created during a assembly process. Since the text error feedback can now also be handled with pipes to the calling application, it is possible to obtain this feedback without an explicit file. The name of the listing file is controlled by the environment variable ERRORFILE.

Example	`-WOutFileOn`
	The error file is created as specified with ERRORFILE.
	`-WErrFileOff`
	No error file is created.

See also	Option -WErrFile Option -WStdout

-WStdout

-WStdout: Write to Standard Output

Group:	MESSAGE
Scope:	Assembly Unit
Syntax:	"-WStdout" ("On" | "Off").
Arguments:	none.
Default:	output is written to stdout.
Description	With Windows applications, the usual standard streams are available. But text written into them does not appear anywhere unless explicitly requested by the calling application. With this option is can be controlled if the text to error file should also be written into the stdout.
Example	

```
-WStdoutOn
```

All messages are written to stdout.

```
-WErrFileOff
```

Nothing is written to stdout.

See also Option -WErrFile
Option -WOutFile

Directive

Assembler directives are described in the *Assembler Directives* chapter.

Sections

Sections are portions of code or data, which cannot be split into smaller element. Each section has a name, a type and some attributes.

Each assembly source file contains at least one section. The number of sections in an assembly source file is only limited by the amount of memory available on the system at assembly time. If inside of a single source file, several sections with the same name are detected, the code is concatenated in one large section.

Sections from different modules, but with the same name will be combined in a single section at linking time.

Each section is defined trough an attribute and a type.

Section Attribute

According to their content each section an attribute is associated with each section. A section may be:

- a data section
- a constant data section
- a code section.

Code Sections

A section containing at least an instruction is considered to be a code section. Code sections are always allocated in the target processor ROM area.

Code sections should not contain any variable definition (variable defined using the DS directive). You will not have any write access on variables defined in a code section. Additionally, these variables cannot be displayed in the debugger as data.

Constant Sections

A section containing only constant data definition (variables defined using the DC or DCB directives) is considered to be a constant section. Constant sections should be allocated in the target processor ROM area, otherwise they cannot be initialized at application loading time.

We strongly recommend you to define separate sections for the definition of variables and constant variables. This will avoid any problems in the initialization of

constant variables.

Data Sections

A section containing only variables (variable defined using the DS directive) is considered to be a data section. Data sections are always allocated in the target processor RAM area.

Note: A section containing variables (DS) and constants (DC) or code is not a data section. Such a section with mixed content is put in ROM by default.

We strongly recommend to define separate sections for the definition of variables and constant variables. This will avoid any problems in the initialization of constant variables.

Section Type

First of all a programmer should decide whether he wants to use relocatable or absolute code in his application. The assembler allows to mix usage of absolute and relocatable sections in a single application and also in a single source file. The main difference between absolute and relocatable sections is the way symbol addresses are determined.

Absolute Section

Relocatable Section

Absolute Sections

The starting address of an absolute section is known at assembly time. An absolute section is defined trough the directive ORG. The operand specified in the ORG directive determines the start address of the absolute section.

Example

```
      XDEF entry
      ORG $A00     ; Absolute constant data section.
cst1: DC.B    $A6
cst2: DC.B    $BC
...
      ORG $800    ; Absolute data section.
var:  DS.B    1
```

```
      ORG $C00    ; Absolute code section.
entry:
      LDAA cst1   ; Load value in cst1
      ADDA cst2   ; Add value in cst2
      STAA var    ; Store in var
      BRA entry
```

In the above example, two bytes of storage are allocated starting at address $A00. Symbol ‘cst1’ will be allocated at address $A00 and ‘cst2’ will be allocated at address $A01. All subsequent instructions or data allocation directives will be located in the absolute section until another section is specified using the ORG or SECTION directive.

When using absolute sections, it is the user responsibility to ensure that there is no overlap between the different absolute sections defined in his application. In the previous example, the programmer should ensure that the size of the section starting at address $A00 is not bigger than $200 bytes, otherwise section starting at $A00 and section starting at $C00 will overlap.

When object files are generated, even applications containing only absolute sections must be linked. In that case, there should not be any overlap between the address ranges from the absolute sections defined in the assembly file and the address ranges defined in the linker parameter file.

Example

The PRM file used to assemble the example above, can be defined as follows:

```
LINK test.abs  /* Name of the executable file generated. */
NAMES
  test.o       /* Name of the object files in the application. */
END
SECTIONS
/* READ_ONLY memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */
  MY_ROM  = READ_ONLY  0x1000 TO 0x1FFF;
/* READ_WRITE memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */
  MY_RAM  = READ_WRITE 0x2000 TO 0x2FFF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  DEFAULT_RAM    INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
```

```
  DEFAULT_ROM    INTO MY_ROM;
END
INIT entry              /*  Application entry point. */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

The linker PRM file contains at least:

- The name of the absolute file (command LINK).
- The name of the object file which should be linked (command NAMES).
- The specification of a memory area where the sections containing variables must be allocated. At least the predefined section DEFAULT_RAM (or its ELF alias '.data') must be placed there. For applications containing only absolute sections, nothing will be allocated there (commands SECTIONS and PLACEMENT).
- The specification of a memory area where the sections containing code or constants must be allocated. At least the predefined section DEFAULT_ROM (or its ELF alias '.text') must be placed there. For applications containing only absolute sections, nothing will be allocated there (commands SECTIONS and PLACEMENT).
- The specification of the application entry point (command INIT)
- The definition of the reset vector (command VECTOR ADDRESS)

Relocatable Sections

The starting address of a relocatable section is evaluated at linking time, according to the information stored in the linker parameter file. A relocatable section is defined trough the directive SECTION.

Example

```
          XDEF entry
constSec: SECTION ; Relocatable constant data section.
cst1:     DC.B    $A6
cst2:     DC.B    $BC
...
dataSec:  SECTION ; Relocatable data section.
var:      DS.B    1

codeSec:  SECTION ; Relocatable code section.
entry:
          LDAA cst1      ; Load value in cst1
          ADDA cst2      ; Add value in cst2
          STAA var       ; Store in var
          BRA entry
```

In the previous example, two bytes of storage are allocated in section ‘constSec’. Symbol ‘cst1’ will be allocated at offset 0 and ‘cst2’ at offset 1 from the beginning of the section. All subsequent instructions or data allocation directives will be located in the relocatable section ‘constSec’ until another section is specified using the ORG or SECTION directive.

When using relocatable sections, the user do not need to care about overlapping sections. The linker will assign a start address to each section according to the input from the linker parameter file.

The customer can decide to define only one memory area for the code and constant sections and another one for the variable sections or to split his sections over several memory area.

Example: Defining one RAM and one ROM Area.

When all constant and code sections as well as data sections can be allocated consecutively, the PRM file used to assemble the example above, can be defined as follows:

```
LINK test.abs  /* Name of the executable file generated. */
NAMES
  test.o       /* Name of the object files in the application. */
END
SECTIONS
/* READ_ONLY memory area.  */
  MY_ROM  = READ_ONLY  0x0B00 TO 0x0BFF;
/* READ_WRITE memory area. */
  MY_RAM  = READ_WRITE 0x0800 TO 0x08FF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  DEFAULT_RAM    INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  DEFAULT_ROM    INTO MY_ROM;
END
INIT entry              /*  Application entry point. */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

The linker PRM file contains at least:

- The name of the absolute file (command LINK).
- The name of the object file which should be linked (command NAMES).

- The specification of a memory area where the sections containing variables must be allocated At least the predefined section DEFAULT_RAM (or its ELF alias '.data') must be placed there. (commands SECTIONS and PLACEMENT).
- The specification of a memory area where the sections containing code or constants must be allocated. At least the predefined section DEFAULT_ROM (or its ELF alias '.text') must be placed there. (commands SECTIONS and PLACEMENT).
- The specification of the application entry point (command INIT)
- The definition of the reset vector (command VECTOR ADDRESS)

According to the PRM file above,

- the section 'dataSec' will be allocated starting at 0x0800.
- the section 'constSec' will be allocated starting at 0x0B00.
- the section 'codeSec will be allocated next to the section 'constSec'.

Example: Defining multiple RAM and ROM Areas.

When all constant and code sections as well as data sections cannot be allocated consecutively, the PRM file used to assemble the example above, can be defined as follows:

```
LINK test.abs  /* Name of the executable file generated. */
NAMES
  test.o       /* Name of the object files in the application. */
END
SECTIONS
  ROM_AREA_1= READ_ONLY  0xB00 TO 0xB7F; /* READ_ONLY memory area. */
  ROM_AREA_2= READ_ONLY  0xC00 TO 0xC7F;/* READ_ONLY memory area. */
  RAM_AREA_1= READ_WRITE 0x800 TO 0x87F;/* READ_WRITE memory area. */
  RAM_AREA_2= READ_WRITE 0x900 TO 0x97F;/* READ_WRITE memory area. */
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  dataSec              INTO RAM_AREA_2;
  DEFAULT_RAM          INTO RAM_AREA_1;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  constSec             INTO ROM_AREA_2;
  codeSec, DEFAULT_ROM  INTO ROM_AREA_1;
END
INIT entry             /*  Application entry point. */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

The linker PRM file contains at least:

- The name of the absolute file (command LINK).
- The name of the object file which should be linked (command NAMES).
- The specification of memory areas where the sections containing variables must be allocated At least the predefined section DEFAULT_RAM (or its ELF alias '.data') must be placed there (commands SECTIONS and PLACEMENT).
- The specification of memory areas where the sections containing code or constants must be allocated. At least the predefined section DEFAULT_ROM (or its ELF alias '.text') must be placed there. (commands SECTIONS and PLACEMENT).
- The specification of the application entry point (command INIT)
- The definition of the reset vector (command VECTOR ADDRESS)

According to the PRM file above,

- the section 'dataSec' will be allocated starting at 0x0900.
- the section 'constsec will be allocated starting at 0x0C00.
- the section 'codeSec' will be allocated starting at 0x0B00.

Relocatable vs. Absolute Section

Generally we recommend to develop application using relocatable sections.Relocatable sections offers several advantages.

Modularity

An application is more modular when programming can be divided into smaller units called sections. The sections themselves can be distributed among different source files.

Multiple Developers

When an application is split over different files, multiple developers can be involved in the development of the application. In order to avoid major problems when merging the different files, attention must be paid to following items:

- An include file must be available for each assembly source file, containing XREF directives for each exported variables, constants and functions. Additionally, the interface to the function should be described there (parameter passing rules as well as function return value).
- When accessing variables, constants or function from another module, the corresponding include file must be included.

- Variables or constants defined by another developer must always be referenced by their names.
- Before invoking a function implemented in another file, the developer should ensure he respect the function interface (parameters are passed as expected, return value is retrieved correctly).

Early Development

The whole application can be developed before the application memory map is known. Often the definitive application memory map can only be determined once the size required for code and data can be evaluated. The size required for code or data can only be quantified once the major part of the application is implemented. When absolute sections are used, defining the definitive memory map is an iterative process of mapping and remapping the code. The assembly files must be edited, assembled and linked several times. When relocatable sections are used, this can be achieved by editing the PRM file and linking the application.

Enhanced Portability

As the memory map is not the same for all MCU derivatives, using relocatable sections allow to easily port the code for another MCU. When porting relocatable code to another target you only need to link the application again, with the appropriate memory map.

Tracking Overlaps

When using absolute sections, the programmer must ensure there is no overlap between his sections. When using relocatable sections, the programmer do not need to take care about sections overlapping. The label offsets are all evaluated relatively to the beginning of the section. Absolute addresses are determined and assigned by the linker.

Reusability

When using relocatable sections, code implemented to handle a specific I/O device (serial communication device), can be reused in another application without any modification.

Assembler Syntax

An assembler source program is a sequence of source statements. Each source statement is coded on one single line of text and can be:

- a comment line
- a source line

Comment Line

A comment can occupy an entire line to explain the purpose and usage of a block of statements or to describe an algorithm. A comment line contains a semicolon followed by a text. Comments are included in the assembly listing, but are not significant to the assembler.

An empty line is also considered as a comment line.

Example:

```
; This is a comment line
```

Source Line

Each source statement includes one or more of the following four fields:

- a label
- an operation field
- one or several operand
- a comment

Characters on the source line may be either upper or lower case. Directives and instructions are case insensitive, whereas symbols are case sensitive unless option for case insensitivity on label names (-Ci) is activated.

Label Field

The label field is the first field in a source line. A label is a symbol followed by a colon. Labels can include letters (‘A’.. ‘Z’ or ‘a’.. ‘z’), underscores, periods and numbers. The first character must not be a number.

Note: For compatibility with other macro assembler vendor, an identifier starting

on column 1 is considered to be a label, even when it is not terminated by a colon.
When option -MCUasm (Switch ON MCUasm Compatibility Mode) is activated, labels MUST be terminated with a colon. An error message is issued, when a label is not followed by a colon.

Labels are required on assembler directives that define the value of a symbol (SET or EQU). For these directives, labels are assigned the value corresponding to the expression in the operand field.

Labels specified in front of another directive, an instruction or a comment are assigned the value of the location counter in the current section.

Note: When the macro assembler expands macro it generates internal symbols starting with an '_'. Therefore, to avoid potential conflicts, user defined symbols should not begin with an underscore

Note: For the macro assembler, a .B or .W at the end of a label has a specific meaning. Therefore, to avoid potential conflicts, user defined symbols should not end with .B or .W.

Operation Field

The operation field follows the label field and is separated from it by a white space. The operation field must not begin in the first column. An entry in the operation field is one of the following:

- an instruction mnemonic
- a directive name
- a macro name

Instruction

Executable instructions for the M68HC12 processor are defined in the "CPU Reference Manual CPU12RM/AD".

The following table presents an overview of the instruction available:

Instruction	Description
ABA	Add accumulator A and B
ABX	Add accumulator B and register X
ABY	Add accumulator B and register Y

Instruction	Description
ADCA	Add with carry to accumulator A
ADCB	Add with carry to accumulator B
ADDA	Add without carry to accumulator A
ADDB	Add without carry to accumulator B
ADDD	Add without carry to accumulator D
ANDA	Logical AND with accumulator A
ANDB	Logical AND with accumulator B
ANDCC	Logical AND with CCR
ASL	Arithmetic shift left in memory
ASLA	Arithmetic shift left accumulator A
ASLB	Arithmetic shift left accumulator B
ASLD	Arithmetic shift left accumulator D
ASR	Arithmetic shift left in memory
ASRA	Arithmetic shift right accumulator A
ASRB	Arithmetic shift right accumulator B
BCC	Branch if carry clear
BCLR	Clear bits in memory
BCS	Branch if carry Set
BEQ	Branch if equal
BGE	Branch if greater than or equal
BGND	Place in BGND mode
BGT	Branch if greater than
BHI	Branch if higher
BHS	Branch if higher or same
BITA	Logical AND accumulator A and memory
BITB	Logical AND accumulator B and memory
BLE	Branch if Less Than or equal
BLO	Branch if lower (Same as BCS)
BLS	Branch if lower or Same
BLT	Branch if less than

Instruction	Description
BMI	Branch if Minus
BNE	Branch if not equal
BPL	Branch if Plus
BRA	Branch Always
BRCLR	Branch if bit clear
BRN	Branch never
BRSET	Branch if bits set
BSET	Set bits in memory
BSR	Branch subroutine
BVC	Branch if overflow cleared
BVS	Branch if overflow set
CALL	call subroutine in extended memory
CBA	Compare accumulator A and B
CLC	Clear carry bit
CLI	Clear interrupt bit
CLR	Clear memory
CLRA	Clear accumulator A
CLRB	Clear accumulator B
CLV	Clear two’s complement overflow bit
CMPA	Compare memory with accumulator A
CMPB	Compare memory with accumulator B
COM	One’s complement on memory location
COMA	One’s complement on accumulator A
COMB	One’s complement on accumulator B
CPD	Compare accumulator D and memory
CPS	Compare register SP and memory
CPX	Compare register X and memory
CPY	Compare register Y and memory
DAA	Decimal adjust accumulator A
DBEQ	Decrement counter and branch if null

Instruction	Description
DBNE	Decrement counter and branch if not null
DEC	Decrement memory location
DECA	Decrement accumulator A
DECB	Decrement accumulator B
DES	Decrement register SP
DEX	Decrement index register X
DEY	Decrement index register Y
EDIV	Unsigned division 32-bits/16 bits
EDIVS	Signed division 32-bits/16 bits
EMACS	Multiply and accumulate signed
EMAXD	Get maximum of 2 unsigned integer in accumulator D
EMAXM	Get maximum of 2 unsigned integer in memory
EMIND	Get minimum of 2 unsigned integer in accumulator D
EMINM	Get minimum of 2 unsigned integer in memory
EMUL	16-bit * 16-bit multiplication (unsigned)
EMULS	16-bit * 16-bit multiplication (signed)
EORA	Logical XOR with accumulator A
EORB	Logical XOR with accumulator B
ETBL	16-Bit Table Lookup and Interpolate
EXG	Exchange register content
FDIV	16-bit / 16-bits fractional divide
IBEQ	Increment counter and branch if null
IBNE	Increment counter and branch if not null
IDIV	16-bit / 16-bit integer division (unsigned)
IDIVS	16-bit / 16-bit integer division (signed)
INC	Increment memory location
INCA	Increment accumulator A
INCB	Increment accumulator B
INS	Increment register SP
INX	Increment register X

Instruction	Description
INY	Increment register Y
JMP	Jump to label
JSR	Jump to subroutine
LBCC	Long branch if carry clear
LBCS	Long branch if carry Set
LBEQ	Long branch if equal
LBGE	Long branch if greater than or equal
LBGT	Long branch if greater than
LBHI	Long branch if higher
LBHS	Long branch if higher or same
LBLE	Long branch if Less Than or equal
LBLO	Long branch if lower (Same as BCS)
LBLS	Long branch if lower or Same
LBLT	Long branch if less than
LBMI	Long branch if Minus
LBNE	Long branch if not equal
LBPL	Long branch if Plus
LBRA	Long branch Always
LBRN	Long branch never
LBSR	Long branch subroutine
LBVC	Long branch if overflow clear
LBVS	Long branch if overflow set
LDAA	Load accumulator A
LDAB	Load accumulator B
LDD	Load accumulator D
LDS	Load register SP
LDX	Load index register X
LDY	Load index register Y
LEAS	Load SP with effective Address
LEAX	Load X with effective Address

Instruction	Description
LEAY	Load Y with effective Address
LSL	Logical shift left in memory
LSLA	Logical shift left accumulator A
LSLB	Logical shift left accumulator B
LSLD	Logical shift left accumulator D
LSR	Logical shift right in memory
LSRA	Logical shift right accumulator A
LSRB	Logical shift right accumulator B
LSRD	Logical shift right accumulator D
MAXA	Get maximum of 2 unsigned byte in accumulator A
MAXM	Get maximum of 2 unsigned byte in memory
MEM	Membership Function
MINA	Get minimum of 2 unsigned byte in accumulator A
MINM	Get minimum of 2 unsigned byte in memory
MOVB	Memory to memory byte move
MOVW	Memory to memory word move
MUL	8 * 8 bit unsigned multiplication
NEG	2’s complement in memory
NEGA	2’s complement accumulator A
NEGB	2’s complement accumulator B
NOP	No operation
ORAA	Logical OR with accumulator A
ORAB	Logical OR with accumulator B
ORCC	Logical OR with CCR
PSHA	Push register A
PSHB	Push register B
PSHC	Push register CCR
PSHD	Push register D
PSHX	Push register X
PSHY	Push register Y

Instruction	Description
PULA	Pop register A
PULB	Pop register B
PULC	Pop register CCR
PULD	Pop register D
PULX	Pop register X
PULY	Pop register Y
REV	MIN-MAX Rule Evaluation for 8-bits values
REVW	MIN-MAX Rule Evaluation for 16-bits values
ROL	Rotate memory left
ROLA	Rotate accumulator A left
ROLB	Rotate accumulator B left
ROR	Rotate memory right
RORA	Rotate accumulator A right
RORB	Rotate accumulator B right
RTC	Return from CALL
RTI	Return from Interrupt
RTS	return from subroutine
SBA	Subtract accumulator A and B
SBCA	Subtract with carry from accumulator A
SBCB	Subtract with carry from accumulator B
SEC	Set carry bit
SEI	Set interrupt bit
SEV	Set two’s complement overflow bit
SEX	Sign extend into 16 bit register
STAA	Store accumulator A
STAB	Store accumulator B
STD	Store accumulator D
STOP	Stop
STS	Store register SP
STX	Store register X

Instruction	Description
STY	Store register Y
SUBA	Subtract without carry from accumulator A
SUBB	Subtract without carry from accumulator B
SUBD	Subtract without carry from accumulator D
SWI	Software interrupt
TAB	Transfer A to B
TAP	Transfer A to CCR
TBA	Transfer B to A
TBEQ	Test counter and branch if null
TBL	8-Bit Table Lookup and Interpolate
TBNE	Test counter and branch if not null
TFR	Transfer register to register
TPA	Transfer CCR to A
TRAP	Software Interrupt
TST	Test memory for 0 or minus
TSTA	Test accumulator A for 0 or minus
TSTB	Test accumulator B for 0 or minus
TSX	Transfer SP to X
TSY	Transfer SP to Y
TXS	Transfer X to SP
TYS	Transfer Y to SP
WAI	Wait for Interrupt
WAV	Weighted Average Calculation
XGDX	Exchange D with X
XGDY	Exchange D with Y

Directive

Assembler directives are described in the “Assembler Directives” chapter in this manual.

Macro Name

A user-defined macro can be invoked in the assembler source program. This results in the expansion of the code defined in the macro. Definition and usage of macros are described in the “Macros” chapter in this manual.

Operand Field: Addressing Modes

The operand fields, when present, follow the operation field and are separated from it by a white space. When two or more operand subfields appear within a statement, a comma must separate them.

The following addressing mode notations are allowed in the operand field:

Addressing Mode	Notation
Inherent	No operands
Direct	<8-bit address>
Extended	<16-bit address>
Relative	<PC relative, 8-Bit offset> or <PC relative, 16-Bit offset>
Immediate	#<immediate 8-bit expression> or #<immediate 16-bit expression>
Indexed, 5-bit offset	<5-bit offset>, xysp
Indexed, pre-decrement	<3-bit offset>, -xys
Indexed, pre-increment	<3-bit offset>, +xys
Indexed, post-decrement	<3-bit offset>, xys-
Indexed, post-increment	<3-bit offset>, xys+
Indexed, accumulator offset	abd, xysp
Indexed, 9-bit offset	<9-bit offset>, xysp
Indexed, 16-bit offset	<16-bit offset>, xysp
Indexed-Indirect, 16-bit offset	[<16-bit offset>, xysp]
Indexed-Indirect, D accumulator offset	[D, xysp]

In the table above:

- xysp stand for one of the index register X, Y, SP, PC or PCR

- xys stand for one of the index register X, Y or SP
- abd stands for one of the accumulator A, B or D

Inherent

Instructions using this addressing mode have no operands or all operands are stored in internal CPU registers. The CPU do not need to perform any memory access to complete the instruction..

Example

```
NOP    ; Instruction with no operand
CLRA   ; The operand is in the CPU register A
```

Immediate

The opcode contains the value to use with the instruction rather than the address of this value. The character '#' is used to indicate an immediate addressing mode operand.

Example

```
main:     LDAA #$64
          LDX  #$AFE
          BRA  main
```

In this example, the hexadecimal value $64 is loaded in register A.
The size of the immediate operand is implied by the instruction context. The register A is a 8-bit register, so the instruction LDAA expect a 8-bit immediate operand. The register X is a 16-bit register, so the instruction LDX expect a 16-bit immediate operand.

The immediate addressing mode can also be used to refer to the address of a symbol.

Example

```
          ORG $80
var1:     DC.B $45, $67
          ORG $800
main:
          LDX  #var1
          BRA  main
```

In this example, the address of the variable 'var1' ($80) is loaded in register X.

Be careful

One very common programming error is to omit the # character. This cause the

assembler to misinterpret the expression as an address rather than an explicit data.

Example

```
LDAA  $60
```

means load accumulator A with the value stored at address $60.

Direct

The direct addressing mode is used to address operands in the direct page of the memory (location `$0000` to `$00FF`).

This addressing mode is used to access operands in the address range $00 to $FF. Access on this memory range (also called zero page) are faster and require less code than the extended addressing mode (see below). In order to speed up his application a programmer can decide to place the most commonly accessed data in this area of memory.

Example

```
            ORG $50
  data:     DS.B 1

  MyCode:   SECTION
  Entry:
            LDS  #$AFE              ; init Stack Pointer
            LDAA #$01
  main:     STAA data
            BRA main
```

In this example, the value in the register `A` is stored in the variable `data` which is located at address $50.

Example

```
  MyData:   SECTION SHORT
  data1:    DS.B 1
            XREF.B data2
  MyCode:   SECTION
  Entry:
            LDS  #$AFE              ; init Stack Pointer
            LDAA data1
  main:     STAA data2
            BRA main
```

Here data1 is located in a relocatable section. To inform the assembler that this section will be placed in the zero page, the SHORT qualifier after SECTION is used. The label data2 is imported into this code. To inform the assembler that this label can also be used with the direct addressing mode, the directive “XREF.B” is used.

Extended

The extended addressing mode is used to access any memory location in the 64-Kilobyte memory map.

Example

```
            XDEF Entry
            ORG $100
data:       DS.B 1
MyCode:     SECTION
Entry:
            LDS  #$AFE              ; init Stack Pointer
            LDAA #$01
main:       STAA data
            BRA main
```

In this example, the value in the register `A` is stored in the variable data. This variable is located at address `$0100` in the memory map.

Relative

This addressing mode is used to determine the destination address of branch instructions. Each conditional branch instruction tests some bits in the condition code register. If the bits are in the expected state, the specified offset is added to the address of the instruction following the branch instruction, and execution continues at that address.

Short branch instructions (BRA, BEQ, ...) expect a signed offset encoded on one byte. The valid range for a short branch offset is [-128..127].

Example

```
main:
        NOP
        NOP
        BRA main
```

In this example, after the two NOPs have been executed, the application branches on the first NOP and continues execution.

Long branch instructions (LBRA, LBEQ, ...) expect a signed offset encoded on two bytes. The valid range for a long branch offset is [-32768..32767].

Using the special symbol for location counter, you can also specify a offset to the location pointer as target for a branch instruction. The * refer to the beginning of the instruction where it is specified.

Example

```
main:
        NOP
        NOP
        BRA *-2
```

In this example, after the two NOPs have been executed, the application branches at offset -2 from the BRA instruction (i.e. on label ‘main’).

Inside of an absolute section, expressions specified in a PC relative addressing mode may be:

- a label defined in any absolute section
- a label defined in any relocatable section
- an external label (defined in a XREF directive)
- an absolute EQU or SET label.

Inside of a relocatable section, expressions specified in a PC relative addressing mode may be:

- a label defined in any absolute section
- a label defined in any relocatable section
- an external label (defined in a XREF directive)

Indexed, 5-bit offset

This addressing mode add a 5-bit signed offset to the base index register to form the memory address, which is referenced in the instruction. The valid range for a 5-bit signed offset is [-16..15]. The base index register may be X, Y, SP, PC or PCR.

For information about Indexed PC and Indexed PC Relative addressing mode, see section ‘Indexed PC vs. Indexed PC Relative Addressing Mode’ below.

This addressing mode may be used to access elements in an n-element table, which size is smaller than 16 bytes.

Example

```
            ORG $1000
CST_TBL:    DC.B $5, $10, $18, $20, $28, $30
            ORG $800
DATA_TBL:   DS.B 10
main:
            LDX #CST_TBL
            LDAA 3,X

            LDY #DATA_TBL
            STAA 8, Y
```

The accumulator A is loaded with the byte value stored in memory location `$1003 ($1000 + 3)`.
Then the value of accumulator A is stored at address `$808 ($800 + 8)`.

Indexed, 9-bit offset

This addressing mode add a 9-bit signed offset to the base index register to form the memory address, which is referenced in the instruction. The valid range for a 9-bit signed offset is [-256..255]. The base index register may be X, Y, SP, PC or PCR.

For information about Indexed PC and Indexed PC Relative addressing mode, see section ‘Indexed PC vs. Indexed PC Relative Addressing Mode’ below.

This addressing mode may be used to access elements in an n-element table, which size is smaller than 256 bytes

Example

```
            ORG $1000
CST_TBL:    DC.B  $5, $10, $18, $20, $28, $30, $38, $40, $48
            DC.B $50, $58, $60, $68, $70, $78, $80, $88, $90
            DC.B $98, $A0, $A8, $B0, $B8, $C0, $C8, $D0, $D8
            ORG $800
DATA_TBL:   DS.B 40
main:
            LDX #CST_TBL
            LDAA 20,X

            LDY #DATA_TBL
            STAA 18, Y
```

The accumulator A is loaded with the byte value stored in memory location `$1014 ($1000 + 20)`.
Then the value of accumulator A is stored at address `$812 ($800 + 18)`.

Indexed, 16-bit offset

This addressing mode add a 16-bit offset to the base index register to form the memory address, which is referenced in the instruction. The 16-bit offset may be considered either as signed or unsigned ($FFFF may be considered to be -1 or 65’535). The base index register may be X, Y, SP, PC or PCR.

For information about Indexed PC and Indexed PC Relative addressing mode, see section ‘Indexed PC vs. Indexed PC Relative Addressing Mode’ below.

Example

```
main:
```

```
            LDX #$600
            LDAA $300,X

            LDY #$1000
            STAA $140, Y
```

The accumulator A is loaded with the byte value stored in memory location `$900` (`$600 + $300`).
Then the value of accumulator A is stored at address `$1140 ($1000 + $140)`.

Indexed, Indirect 16-bit offset

This addressing mode add a 16-bit offset to the base index register to form the address of a memory location containing a pointer to the memory location referenced in the instruction. The 16-bit offset may be considered either as signed or unsigned ($FFFF may be considered to be -1 or 65’535). The base index register may be X, Y, SP, PC or PCR.

For information about Indexed PC and Indexed PC Relative addressing mode, see section ‘Indexed PC vs. Indexed PC Relative Addressing Mode’ below.

Example

```
            ORG $1000
CST_TBL1:   DC.W  $1020, $1050, $2001
            ORG $2000
CST_TBL:    DC.B  $10, $35, $46
            ORG $3000
main:
            LDX #CST_TBL1
            LDAA [4,X]
```

The offset ‘4’ is added to the value of register ‘X’ ($1000) to form the address $1004.
Then an address pointer ($2001) is read from memory at $1004.
The accumulator A is loaded with $35, the value stored at address $2001.

Indexed, pre-decrement

This addressing mode allow you to decrement the base register by a specified value, before indexing takes place. The base register is decremented by the specified value and the content of the modified base register is referenced in the instruction.

The valid range for a pre-decrement value is [1..8]. The base index register may be X, Y, SP.

Example

```
            ORG $1000
CST_TBL:    DC.B $5, $10, $18, $20, $28, $30
END_TBL:    DC.B $0
main:
            CLRA
            CLRB
            LDX   #END_TBL
loop:
            ADDD 1,-X
            CPX   #CST_TBL
            BNE   loop
```

The base register X is loaded with the address of the element following the table CST_TBL ($1006).
The register X is decremented by 1 (its value is $1005) and the value at this address ($30) is added to register D.
X is not equal to the address of CST_TBL, so it is decremented again and the content of address ($1004) is added to D.
This loop is repeated as long as the register X did not reach the beginning of the table CST_TBL ($1000).

Indexed, pre-increment

This addressing mode allow you to increment the base register by a specified value, before indexing takes place. The base register is incremented by the specified value and the content of the modified base register is referenced in the instruction.

The valid range for a pre-increment value is [1..8]. The base index register may be X, Y, SP.

Example

```
            ORG $1000
CST_TBL:    DC.B $5, $10, $18, $20, $28, $30
END_TBL:    DC.B $0
main:
            CLRA
            CLRB
            LDX #CST_TBL
loop:
            ADDD 2,+X
            CPX   #END_TBL
            BNE   loop
```

The base register X is loaded with the address of the table CST_TBL ($1000).
The register X is incremented by 2 (its value is $1002) and the value at this address

($18) is added to register D.
X is not equal to the address of END_TBL, so it is incremented again and the content of address ($1004) is added to D.
This loop is repeated as long as the register X did not reach the end of the table END_TBL ($1006).

Indexed, post-decrement

This addressing mode allow you to decrement the base register by a specified value, after indexing takes place. The content of the base register is read and then the base register is decremented by the specified value.

The valid range for a pre-decrement value is [1..8]. The base index register may be X, Y, SP.

Example

```
            ORG $1000
CST_TBL:    DC.B $5, $10, $18, $20, $28, $30
END_TBL:    DC.B $0
main:
            CLRA
            CLRB
            LDX   #END_TBL
loop:
            ADDD  2,X-
            CPX   #CST_TBL
            BNE   loop
```

The base register X is loaded with the address of the element following the table CST_TBL ($1006).
The value at address $1006 ($0) is added to register D and the X is decremented by 2 (its value is $1004).
X is not equal to the address of CST_TBL, so the value at address $1004 is added to D and X is decremented by two again (its value is now $1002).
This loop is repeated as long as the register X did not reach the beginning of the table CST_TBL ($1000).

Indexed, post-increment

This addressing mode allow you to increment the base register by a specified value, after indexing takes place. The content of the base register is read and then the base register is incremented by the specified value.

The valid range for a pre-increment value is [1..8]. The base index register may be

X, Y, SP.

Example

```
            ORG $1000
CST_TBL:    DC.B $5, $10, $18, $20, $28, $30
END_TBL:    DC.B $0
main:
            CLRA
            CLRB
            LDX #CST_TBL
loop:
            ADDD 1,X+
            CPX   #END_TBL
            BNE   loop
```

The base register X is loaded with the address of the table CST_TBL ($1000).
The value at address $1000 ($5) is added to register D and then the register X is incremented by 1 (its value is $1001).
X is not equal to the address of END_TBL, so the value at address $1001 ($10) is added to register D and then the register X is incremented by 1 (its value is $1002).
This loop is repeated as long as the register X did not reach the end of the table END_TBL ($1006).

Indexed, Accumulator offset

This addressing mode add the value in the specified accumulator to the base index register to form the address, which is referenced in the instruction. The base index register may be X, Y, SP or PC. The accumulator may be A, B or D.

Example

```
main:
            LDAB #$20
            LDX  #$600
            LDAA B,X

            LDY #$1000
            STAA $140, Y
```

The value stored in B ($20) is added to the value of X ($600) to form a memory address ($620). The value stored at $620 is loaded in accumulator A.

Indexed-Indirect, D Accumulator offset

This addressing mode add the value in D to the base index register to form the address of a memory location containing a pointer to the memory location refer-

enced in the instruction. The base index register may be X, Y, SP or PC.

Example

```
entry1:     NOP
            NOP
entry2:     NOP
            NOP
entry3:     NOP
            NOP
main:
            LDD  #2
            JMP  [D, PC]
goto1:      DC.W entry1
goto2:      DC.W entry2
goto3:      DC.W entry3
```

This example is an example of jump table. The values beginning at goto1 are potential destination for the jump instruction.
When JMP [D, PC] is executed, PC points to goto1 and D holds the value 2.
The JMP instruction adds the value in D and PC to form the address of goto2.
The CPU reads the address stored there (the address of the label entry2) and jump there.

Indexed PC vs. Indexed PC Relative Addressing Mode

When using the indexed addressing mode with PC as base register, the macro assembler allow you to use either Indexed PC (<offset>, PC) or Indexed PC Relative (<offset>, PCR) notation.

When Indexed PC notation is used, the offset specified in inserted directly in the opcode.

Example

```
main:
            LDAB  3, PC
            DC.B  $20, $30, $40, $50
```

In the example above, the register B is loaded with the value stored at address PC + 3 ($50).

When Indexed PC Relative notation is used, the offset between the current location counter and the specified expression is computed and inserted in the opcode.

Example

```
main:
            LDAB  x4, PCR
```

```
x1:         DC.B  $20
x2:         DC.B  $30
x3:         DC.B  $40
x4:         DC.B  $50
```

In the example above, the register B is loaded with the value at stored at label ‘X4’ ($50). The macro assembler evaluates the offset between the current location counter and the symbol ‘x4’ to determine the value, which must be stored in the opcode.

Inside of an absolute section, expressions specified in an indexed PC relative addressing mode may be:

- a label defined in any absolute section
- a label defined in any relocatable section
- an external label (defined in a XREF directive)
- an absolute EQU or SET label.

Inside of a relocatable section, expressions specified in an indexed PC relative addressing mode may be:

- a label defined in any absolute section
- a label defined in any relocatable section
- an external label (defined in a XREF directive)

Comment Field

The last field in a source statement is an optional comment field. A semicolon (;) is the first character in the comment field.

Example:

```
NOP ; Comment following an instruction
```

Symbols

User Defined Symbols

Symbols identify memory locations in program or data sections in an assembly module. A symbol has two attributes:

- The section, in which the memory location is defined
- The offset from the beginning of that section.

Symbols can be defined with an absolute or relocatable value, depending on the section in which the labeled memory location is found. If the memory location is located within a relocatable section (defined with the SECTION directive), the label has a relocatable value relative to the section start address.

Symbols can be defined relocatable in the label field of an instruction or data definition source line.

Example

```
Sec: SECTION
label1: DC.B 2 ; label1 is assigned offset 0 within Sec.
label2: DC.B 5 ; label2 is assigned offset 2 within Sec.
label3: DC.B 1 ; label3 is assigned offset 7 within Sec.
```

It is also possible to define a label with either an absolute or a previously defined relocatable value, using a SET or EQU directives.

Symbols with absolute values must be defined with constant expressions.

Example

```
Sec: SECTION
label1: DC.B 2     ; label1 is assigned offset 0 within Sec.
label2: EQU 5      ; label2 is assigned value 5.
label3: EQU label1 ; label3 is assigned the address of label1.
```

External Symbols

A symbol may be made external using the XDEF directive. In another source file a XREF directives must reference it. Since its address is unknown in the referencing file, it is considered to be relocatable.

Example

```
        XREF extLabel ; symbol defined in an other module.
                      ; extLabel is imported in the current module
```

```
        XDEF label    ; symbol is made external for other modules
                      ; label is exported from the current module
constSec: SECTION
label:    DC.W 1, extLabel
```

Undefined Symbols

If a label is neither defined in the source file, nor declared external using XREF, the assembler considers it to be undefined and generates an error.

Example:

```
codeSec: SECTION
entry:
    NOP
    BNE  entry
    NOP
    JMP  end
    JMP  label  ; <- Undeclared user defined symbol: label
end:RTS
    END
```

Reserved Symbols

Reserved symbols cannot be used for user defined symbols.

Register names are reserved identifiers.

For the HC12 processor these reserved identifiers are:

A, B, CCR, D, X, Y, SP, PC, PCR, TEMP1, TEMP2.

Additionally, the keywords **HIGH**, **LOW** and **PAGE** are also a reserved identifier. It is used to refer to the bits 16-23 of a 24-bit value.

Constants

The assembler supports integer and ASCII string constants:

Integer Constants

The assembler supports four representations of integer constants:

- A decimal constant is defined by a sequence of decimal digits (0-9).
  Example 5, 512, 1024
- A hexadecimal constant is defined by a dollar character ($) followed by a sequence of hexadecimal digits (0-9, a-f, A-F).
  Example $5, $200, $400
- An octal constant is defined by the commercial at character (@) followed by a sequence of octal digits (0-7).
  Example @5, @1000, @2000
- A binary constant is defined by a percent character followed by a sequence of binary digits (0-1).
  Example %101, %1000000000, %10000000000

The default base for integer constant is initially decimal, but it can be changed using the BASE directive. When the default base is not decimal, decimal values cannot be represented, because they do not have a prefix character.

String Constants

A string constant is a series of printable characters enclosed in single (‘) or double quote (“). Double quotes are only allowed within strings delimited by single quotes. Single quotes are only allowed within strings delimited by double quotes.

Example

```
'ABCD', "ABCD", 'A', "'B", "A'B", 'A"B'
```

Floating-Point Constants

The macro assembler does not support floating-point constants.

Operators

Operators recognized by the assembler in expressions are:

Addition and Subtraction Operators (binary)

Syntax

```
Addition:    <operand> + <operand>
Subtraction: <operand> – <operand>.
```

Description

The + operator adds two operands, whereas the – operator subtracts them. The operands can be any expression evaluating to an absolute or relocatable expression.

Addition between two relocatable operands is not allowed.

Example

```
$A3216 + $42       ; Addition of two absolute operands ( = $A3258).
label - $10        ; Subtraction with value of ‘label’
```

Multiplication, Division and Modulo Operators (binary)

Syntax

```
Multiplication: <operand> * <operand>
Division:       <operand> / <operand>
Modulo:         <operand> % <operand>
```

Description

The * operator multiplies two operands, the / operator performs an integer division of the two operands and returns the quotient of the operation. The % operator performs an integer division of the two operands and returns the remainder of the operation

The operands can be any expression evaluating to an absolute expression. The second operand in a division or modulo operation cannot be zero.

Example

```
23 * 4    ; multiplication ( = $92).
23 / 4    ; division ( = 5).
23 % 4    ; remainder( = 3).
```

Sign Operators (unary)

Syntax

```
Plus:   +<operand>
Minus:  -<operand>
```

Description

The + operator do not change the operand, whereas the – operator changes the oper-

and to its two's complement. These operators are valid for absolute expression operands.

Example

```
+$32       ; ( = $32).
-$32       ; ( = $CE = -$32).
```

Shift Operators (binary)

Syntax

```
Shift left:  <operand> << <count>
Shift right: <operand> >> <count>
```

Description

The << operator shifts its left operand left by the number of bytes specified in the right operand.

The >> operator shifts its left operand right by the number of bytes specified in the right operand.

The operands can be any expression evaluating to an absolute expression.

Example

```
$25 << 2    ; shift left ( = $94).
$A5 >> 3    ; shift right( = $14).
```

Bitwise Operators (binary)

Syntax

```
Bitwise AND:       <operand> &  <operand>
Bitwise OR:        <operand> | <operand>
Bitwise XOR:       <operand> ^  <operand>
```

Description

The & operator performs an AND between the two operands on bit level.

The | operator performs an OR between the two operands on bit level.

The ^ operator performs a XOR between the two operands on bit level.

The operands can be any expression evaluating to an absolute expression.

Example

```
$E & 3      ; = $2 (%1110 & %0011 = %0010)
$E | 3      ; = $F (%1110 | %0011 = %1111)
$E ^ 3      ; = $D (%1110 ^ %0011 = %1101)
```

Bitwise Operators (unary)

Syntax

```
One's complement: ~<operand>
```

Description

The ~ operator evaluates the one's complement of the operand.

The operand can be any expression evaluating to an absolute expression.

Example

```
~$C         ; = $FFFFFFF3 (~%00000000 00000000 00000000 00001100
                           =%11111111 11111111 11111111 11110011)
```

Logical Operators (unary)

Syntax

```
Logical NOT: !<operand>
```

Description

The ! operator returns 1 (true) if the operand is 0, otherwise it returns 0 (false).

The operand can be any expression evaluating to an absolute expression.

Example

```
!(8<5)      ; = $1 (TRUE)
```

Relational Operators (binary)

Syntax

```
Equal:              <operand> =  <operand>
                    <operand> == <operand>
Not equal:          <operand> != <operand>
```

```
                      <operand> <> <operand>
Less than:            <operand> <  <operand>
Less than or equal:   <operand> <= <operand>
Greater than:         <operand> >  <operand>
Greater than or equal:<operand> >= <operand>
```

Description

These operators compares the two operands and return 1 if the condition is ‘true’ or 0 if the condition is ‘false’.

The operands can be any expression evaluating to an absolute expression.

Example

```
3 >= 4      ; = 0  (FALSE)
label = 4   ; = 1  (TRUE) if label is 4, 0 (FALSE) otherwise.
9 <  $B     ; = 1  (TRUE)
```

HIGH Operator

Syntax

```
High Byte:  HIGH(<operand>)
```

Description

This operator returns the high byte of the address of a memory location.

Example:

Assume `data1` is a word located at address `$1050` in the memory.

```
LDAA #HIGH(data1)
```

This instruction will load the immediate value of the high byte of the address of `data1` (`$10`) in register A.

```
LDAA HIGH(data1)
```

This instruction will load the direct value at memory location of the higher byte of the address of `data1` (i.e. the value in memory location `$10`) in register A.

LOW Operator

Syntax

```
LOW Byte:  LOW(<operand>)
```

Description

This operator returns the low byte of the address of a memory location.

Example:

Assume `data1` is a word located at address `$1050` in the memory.

```
LDAA  #LOW(data1)
```

This instruction will load the immediate value of the lower byte of the address of `data1` (`$50`) in register A.

```
LDAA  LOW(data1)
```

This instruction will load the direct value at memory location of the lower byte of the address of `data1` (i.e. the value in memory location `$50`) in register A.

PAGE Operator

Syntax

```
PAGE Byte:  PAGE(<operand>)
```

Description

This operator returns the page byte of the address of a memory location.

Example:

Assume `data1` is a word located at address `$28050` in the memory.

```
LDAA #PAGE(data1)
```

This instruction will load the immediate value of the page byte of the address of `data1` (`$2`).

```
LDAA PAGE(data1)
```

This instruction will load the direct value at memory location of the page byte of the address of `data1` (i.e. the value in memory location `$2`).

Force Operator (unary)

Syntax

```
8-bit address:  <<operand>
                <operand>.B
16-bit address: ><operand>
                <operand>.W
```

Description

The < or .B operators force the operand to be an 8-bit operand, whereas the > or .W operators force the operand to be a 16-bit operand.

< operator may be useful to force the 8-bit immediate, 8-bit indexed or direct addressing mode for an instruction.

> operator may be useful to force the 16-bit immediate, 16-bit indexed or extended addressing mode for an instruction.

The operand can be any expression evaluating to an absolute or relocatable expression.

Example:

```
<label        ; label is a 8-bit address.
label.B       ; label is a 8-bit address.
>label        ; label is a 16-bit address.
label.W       ; label is a 16-bit address.
```

Operator Precedence

Operator precedence follows the rules for ANSI - C operators.

Operator	Description	Associativity
()	Parenthesis	Right to Left
~ + -	One's complement Unary Plus Unary minus	Left to Right
* / %	Integer multiplication Integer division Integer modulo	Left to Right

Operator	Description	Associativity
+ -	Integer addition Integer subtraction	Left to Right
<< >>	Shift Left Shift Right	Left to Right
< <= > >=	Less than Less or equal to Greater than Greater or equal to	Left to Right
=, == !=, <>	Equal to Not Equal to	Left to Right
&	Bitwise AND	Left to Right
^	Bitwise Exclusive OR	Left to Right
|	Bitwise OR	Left to Right

Expression

An expression is composed of one or more symbols or constants, which are combined with unary or binary operators. Valid symbols in expressions are:

- User defined symbols
- External symbols
- The special symbol '*' represents the value of the location counter at the beginning of the instruction or directive, even when several arguments are specified. In the following example, the asterisk represents the location counter at the beginning of the DC directive:

```
DC.W   1, 2, *-2
```

Once a valid expression has been fully evaluated by the assembler, it is reduced as one of the following type of expressions:

- **Absolute expression**: The expression has been reduced to an absolute value, which is independent of the start address of any relocatable section. Thus it is a constant.
- **Simple relocatable expression**: The expression evaluates to an absolute offset from the start of a single relocatable section.
- **Complex relocatable expression**: The expression neither evaluates to an absolute expression nor to a simple relocatable expression. The assembler does not

support such expressions.

All valid user defined symbols representing memory locations are simple relocatable expressions. This includes labels specified in `XREF` directives, which are assumed to be relocatable symbols.

Absolute Expression

An absolute expression is an expression involving constants or known absolute labels or expressions . An expression containing an operation between an absolute expression and a constant value is also an absolute expression.

Example of absolute expression:

```
Base:  SET $100
Label: EQU Base * $5 + 3
```

Expressions involving the difference between two relocatable symbols defined in the same file and in the same section evaluate to an absolute expression. An expression as "`label2-label1`" can be translated as:

```
(<offset label2> + <start section address >) –
(<offset label1> + <start section address >)
```

This can be simplified as:

```
<offset label2> + <start section address > –
<offset label1> - <start section address>
= <offset label2> - <offset label1>
```

Example

In the following example the expression "`tabEnd-tabBegin`" evaluates to an absolute expression, and is assigned the value of the difference between the offset of `tabEnd` and `tabBegin` in the section `DataSec`.

```
DataSec:  SECTION
tabBegin: DS.B  5
tabEnd:   DS.B  1

ConstSec: SECTION
label:    EQU tabEnd-tabBegin     ; Absolute expression

CodeSec:  SECTION
entry:    NOP
```

Simple Relocatable Expression

A simple relocatable expression results from an operation like:

- <relocatable expression> + <absolute expression>
- <relocatable expression> - <absolute expression>
- < absolute expression> + < relocatable expression>

Example

```
          XREF XtrnLabel
DataSec: SECTION
tabBegin: DS.B  5
tabEnd:   DS.B  1
CodeSec: SECTION
entry:
          LDAA tabBegin+2       ; Simple relocatable expression
          BRA  *-3              ; Simple relocatable expression
          LDAA XtrnLabel+6      ; Simple relocatable expression
```

Unary Operation Result

The following table describes the type of an expression according to the operator in an unary operation:

Operator	Operand	Expression
-, !, ~	absolute	absolute
-, !, ~	relocatable	complex
+	absolute	absolute
+	relocatable	relocatable

Binary Operations Result

The following table describes the type of an expression according to the left and right operators in a binary operation:

Operator	Left Operand	Right Operand	Expression
-	absolute	absolute	absolute

Operator	Left Operand	Right Operand	Expression
-	relocatable	absolute	relocatable
-	absolute	relocatable	complex
-	relocatable	relocatable	absolute
+	absolute	absolute	absolute
+	relocatable	absolute	relocatable
+	absolute	relocatable	relocatable
+	relocatable	relocatable	complex
*, /, %, <<, >>, |, &, ^	absolute	absolute	absolute
*, /, %, <<, >>, |, &, ^	relocatable	absolute	complex
*, /, %, <<, >>, |, &, ^	absolute	relocatable	complex
*, /, %, <<, >>, |, &, ^	relocatable	relocatable	complex

Translation Limits

The following limitations apply to the macro assembler:

- Floating-point constants are not supported.
- Complex relocatable expressions are not supported.
- Lists of operands or symbols must be separated with a comma.
- Include may be nested up to 50.
- The maximum line length is 1023.

Assembler Directives

There are different class of assembler directives. The following tables gives you an overview over the different directives and their class:

Directive Overview

Section Definition Directives

These directives are used to define new sections.

Directive	Description
ORG	Define an absolute section
SECTION	Define a relocatable section
OFFSET	Define an offset section

Constant Definition Directives

These directives are used to define assembly constants.

Directive	Description
EQU	Assign a name to an expression (cannot be redefined)
SET	Assign a name to an expression (can be redefined)

Data Allocation Directives

These directives are used to allocate variables.

Directive	Description
DC	Define a constant variable
DCB	Define a constant block
DS	Define storage for a variable
RAD50	RAD50 encoded string constants

Symbol Linkage Directives

These directives are used to export or import global symbols.

Directive	Description
ABSENTRY	Specify the application entry point when an absolute file is generated
XDEF	Make a symbol public (Visible from outside)
XREF	Import reference to an external symbol.
XREFB	Import reference to an external symbol located on the direct page.

Assembly Control Directives

These directives are general purpose directives used to control the assembly process.

Directive	Description
ALIGN	Define Alignment Constraint
BASE	Specify default base for constant definition
END	End of assembly unit
ENDFOR	End of FOR block
EVEN	Define 2 Byte alignment constraint
FAIL	Generate user defined error or warning messages
FOR	Repeat assembly blocks
INCLUDE	Include text from another file.
LONGEVEN	Define 4 Byte alignment constraint

Listing File Control Directives

These directives controls the generation of the assembler listing file.

Directive	Description
CLIST	Specify if all instructions in a conditional assembly block must be inserted in the listing file or not.
LIST	Specify that all subsequent instructions must be inserted in the listing file.
LLEN	Define line length in assembly listing file.
MLIST	Specify if the macro expansions must be inserted in the listing file.
NOLIST	Specify that all subsequent instruction must not be inserted in the listing file.
NOPAGE	Disable paging in the assembly listing file.
PAGE	Insert page break.
PLEN	Define page length in the assembler listing file.
SPC	Insert an empty line in the assembly listing file.
TABS	Define number of character to insert in the assembler listing file for a TAB character.
TITLE	Define the user defined title for the assembler listing file.

Macro Control Directives

These directives are used for the definition, expansion of macros.

Directive	Description
ENDM	End of user defined macro.
MACRO	Start of user defined macro.
MEXIT	Exit from macro expansion.

Conditional Assembly Directives

These directives are used for conditional assembling.

Directive	Description
ELSE	alternate block
ENDIF	End of conditional block
IF	Start of conditional block. A boolean expression follows this directive.
IFC	Test if two string expressions are equal.
IFDEF	Test if a symbol is defined.
IFEQ	Test if an expression is null.
IFGE	Test if an expression is greater or equal to 0.
IFGT	Test if an expression is greater than 0.
IFLE	Test if an expression is less or equal to 0.
IFLT	Test if an expression is less than 0.
IFNC	Test if two string expressions are different.
IFNDEF	Test if a symbol is undefined
IFNE	Test if an expression is not null.

ABSENTRY - Application Entry Point

Syntax:

```
ABSENTRY <label>
```

Synonym:

None

Description

This directive allow to specify the application Entry Point when the assembler generates directly an absolute file (the option -FA2 ELF/DWARF 2.0 Absolute File must be enabled).

Using this directive, the entry point of the assembly application is written in the header of the generated absolute file. When this file is loaded in the debugger, the line where the entry point label is defined is highlighted in the source window.

This directive is ignored, when the assembler generates an object file.

Note: This instruction does only affect the loading on an application by a debugger. It tells the debugger which initial PC should be used. In order to start the application on a target, initialize the reset vector.

Example

If the example below is assembled using the -FA2 option, an Elf/Dwarf 2.0 Absolute file is generated.

```
       ABSENTRY entry

       ORG $fffe
Reset: DC.W entry
       ORG $70
entry: NOP
       NOP
main:  LDS #$1FFF
       NOP
       BRA main
```

According to the ABSENTRY directive, the Entry Point will be set to the address of entry in the header of the absolute file.

ALIGN - Align Location Counter

Syntax:

`ALIGN <n>`

Synonym:

None

Description

This directive forces the next instruction to a boundary that is a multiple of <n>, relative to the start of the section. The value of <n> must be a positive number between 1 and 32767. The `ALIGN` directive can force alignment to any size. The filling bytes inserted for alignment purpose are initialized with '\0'.

`ALIGN` can be used in code or data sections.

Example

The following example aligns the *HEX* label to a location, which is a multiple of 16 (in this case, location `00010` (Hex))

```
Assembler

Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
    1    1
    2    2   000000 6869 6768         DC.B "high"
    3    3   000004 0000 0000         ALIGN 16
             000008 0000 0000
             00000C 0000 0000
    4    4
    5    5
    6    6   000010 7F           HEX:  DC.B 127  ; HEX is allocated
    7    7                                       ; on an address,
    8    8                                       ; which is a
    9    9                                       ; multiple of 16.
```

BASE - Set Number Base

Syntax:

BASE <n>

Synonym:

None

Description

The directive sets the default number base for constants to <n>. The operand <n> may be prefixed to indicate its number base; otherwise, the operand is considered to be in the current default base. Valid values of <n> are 2, 8, 10, 16. Unless a default base is specified using the BASE directive, the default number base is decimal.

Example

```
 4    4                    base    10   ; default base: decimal
 5    5   000000 64        dc.b    100
 6    6                    base    16   ; default base: hex.
 7    7   000001 0A        dc.b    0a
 8    8                    base    2    ; default base: binary
 9    9   000002 04        dc.b    100
10   10   000003 04        dc.b    %100
11   11                    base    @12  ; default base: decimal
12   12   000004 64        dc.b    100
13   13                    base    $a   ; default base: decimal
14   14   000005 64        dc.b    100
15   15
16   16                    base    8    ; default base: octal
17   17   000006 40        dc.b    100
```

Be careful

Even if the base value is set to 16, hexadecimal constants terminated by a ‘D’ must be prefixed by the $ character, otherwise they are supposed to be decimal constants in old style format. For example, constant 45D is interpreted as decimal constant 45, not as hexadecimal constant 45D.

CLIST - List Conditional Assembly

Syntax:

```
CLIST [ON | OFF]
```

Synonym:

None

Description

The `CLIST` directive controls the listing of subsequent conditional assembly blocks. It precedes the first directive of the conditional assembly block to which it applies, and remains effective until the next `CLIST` directive is read.

When the `ON` keyword is specified in a `CLIST` directive, the listing file includes all directives and instructions in the conditional assembly block, even those which do not generate code (which are skipped).

When the `OFF` keyword is entered, only the directives and instructions that generates code are listed.

A soon as the option –L is activated, the assembler defaults to `CLIST ON`.

Example

Listing file with CLIST OFF

```
         CLIST OFF
Try:     EQU   0
         IFEQ  Try
           LDAA #103
         ELSE
           LDAA #0
         ENDIF
```

The corresponding listing file is:

```
  Abs. Rel.   Loc    Obj. code   Source line
  ---- ----   ------ ---------   -----------
     2    2          0000 0000   Try:    EQU   0
     3    3          0000 0000           IFEQ  Try
     4    4   000000 8667                  LDAA #103
     5    5                              ELSE
     7    7                              ENDIF
```

Listing file with `CLIST ON`

When assembling the code:

```
        CLIST ON
Try:    EQU   0
        IFEQ  Try
          LDAA #103
        ELSE
          LDAA #0
        ENDIF
```

The corresponding listing file is:

```
HC12-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    2    2          0000 0000   Try:    EQU   0
    3    3          0000 0000           IFEQ  Try
    4    4   000000 8667                  LDAA #103
    5    5                               ELSE
    6    6                                 LDAA #0
    7    7                               ENDIF
```

DC - Define Constant

Syntax:

```
[<label>:]  DC [.<size>] <expression> [, <expression>]...
```

where <size> = B (default), W or L.

Synonym:

```
DCW (= 2 byte DC's), DCL (= 4 byte DC's), FCB (= DC.B), FDB (== 2 byte
DC's), FQB (= 4 byte DC's)
```

Description

The DC directive defines constants in memory. It can have one or more <expression> operands, which are separated by commas. The <expression> can contain an actual value (binary, octal, decimal, hexadecimal or ASCII). Alternatively, the <expression> can be a symbol or expression that can be evaluated by the assembler as an absolute or simple relocatable expression. One memory block is allocated and initialized for each expression.

The following rules apply to size specifications for DC directives:

- **DC.B**: One byte is allocated for numeric expressions. One byte is allocated per ASCII character for strings.
- **DC.W**: Two bytes are allocated for numeric expressions. ASCII strings are right aligned on a two-byte boundary.
- **DC.L**: Four bytes are allocated for numeric expressions. ASCII strings are right aligned on a four byte boundary

Example for DC.B:

```
000000 4142 4344   Label: DC.B "ABCDE"
000004 45
000005 0A0A 010A          DC.B %1010, @12, 1,$A
```

Example for DC.W:

```
000000 0041 4243   Label: DC.W "ABCDE"
000004 4445
000006 000A 000A          DC.W %1010, @12, 1, $A
00000A 0001 000A
```

```
00000E xxxx              DC.W Label
```

Example for DC.L:

```
000000 0000 0041   Label: DC.L "ABCDE"
000004 4243 4445
000008 0000 000A         DC.L %1010, @12, 1, $A
00000C 0000 000A
000010 0000 0001
000014 0000 000A
000018 xxxx xxxx         DC.L Label
```

If the value in an operand expression exceeds the size of the operand, the value is truncated and a warning message is generated.

See also

SECTION Directive

ORG Directive

DCB Directive

DS Directive

DCB - Define Constant Block

Syntax:

```
[<label>:]  DCB [.<size>] <count>, <value>
```

where <size> = `B` (default), `W` or `L`.

Description

The `DCB` directive causes the assembler to allocate a memory block initialized with the specified `<value>`. The length of the block is `<size> * <count>`.

`<count>` may not contain undefined, forward, or external references. It may range from 1 to 4096.

The value of each storage unit allocated is the sign-extended expression `<value>`, which may contain forward references. The `<count>` cannot be relocatable. This directive does not perform any alignment.

The following rules apply to size specifications for `DCB` directives:

- **DCB.B**: One byte is allocated for numeric expressions.
- **DCB.W**: Two bytes are allocated for numeric expressions.
- **DCB.L**: Four bytes are allocated for numeric expressions.

Example

```
000000 FFFF FF     Label: DCB.B 3, $FF
000003 FFFE FFFE          DCB.W 3, $FFFE
000007 FFFE
000009 0000 FFFE          DCB.L 3, $FFFE
00000D 0000 FFFE
000011 0000 FFFE
```

See also

SECTION Directive

ORG Directive

DC Directive

DS Directive

DS - Define Space

Syntax:

```
[<label>:]  DS [.<size>]  <count>
```

where <size> = B (default), W or L.

Synonym:

```
RMB (= DS.B)
RMD (2 bytes)
RMQ (4 bytes)
```

Description

The DS directive is used to reserve memory for variables. The content of the memory reserved is not initialized. The length of the block is `<size> * <count>`.

`<count>` may not contain undefined, forward, or external references. It may range from 1to 4096.

Example

```
Counter:  DS.B  2 ; 2 continuous bytes in memory
          DS.B  2 ; 2 continuous bytes in memory
                  ; can only be accessed trough the label Counter
          DS.W  5 ; 5 continuous words in memory
```

The label '`Counter`' references the lowest address of the defined storage area.

Note: Storage allocated with a DS directive may end up in constant data section or even in a code section, if the same section contains constants or code as well. The assembler allocates only a complete section at once.
Example:

```
; How it should NOT be done ...
Counter:        DS 1     ; 1 byte space
InitialCounter: DC.B $f5 ; constant $f5
main:           NOP      ; NOP instruction
```

In the example code above, a variable, a constant and code are put into the same section. Because code has to be in ROM, all 3 elements are put into ROM. In order to allocate them separately, put them in different sections:

```
; How it should be done ...
DataSecti:      SECTION  ; section for variables
Counter:        DS 1     ; 1 byte space

ConstSect:      SECTION  ; section for constants
InitialCounter: DC.B $f5 ; constant $f5

CodeSect:       SECTION  ; section for code
main:           NOP      ; NOP instruction
```

An ORG directive does also start a new section.

See also

SECTION Directive

ORG Directive

DC Directive

ELSE - Conditional Assembly

Syntax:

```
IF  <condition>
  [<assembly language statements>]
[ELSE]
  [<assembly language statements>]
ENDIF
```

Synonym:

```
ELSEC
```

Description

If `<condition>` is true, the statements between `IF` and the corresponding `ELSE` directive are assembled (generate code).

If `<condition>` is false, the statements between `ELSE` and the corresponding `ENDIF` directive are assembled. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

Example

The following is an example of the use of conditional assembly directives:

```
Try: EQU 1
     IF  Try != 0
       LDAA   #103
     ELSE
       LDAA   #0
     ENDIF
```

The value of *Try* determines the instruction to be assembled in the program. As shown, the "*ldaa #103"* instruction is assembled. Changing the operand of the "*equ"* directive to one causes the "*ldaa #0"* instruction to be assembled instead. The following shows the listing provided by the assembler for these lines of code:

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
    1    1          0000 0001   Try: EQU 1
    2    2          0000 0001        IF  Try != 0
    3    3   000000 8667               LDAA   #103
    4    4                        ELSE
```

```
        6    6                          ENDIF
```

END - End Assembly

Syntax:

```
END
```

Synonym:

None

Description

The `END` directive indicates the end of the source code. Subsequent source statements in this file are ignored. The `END` directive in included files skips only subsequent source statements in this include file. The assembly continues in the including file in a regular way.

Example

Source File

```
Label:  DC.W    $1234
        DC.W    $5678
        END
        DC.W    $90AB ; no code generated
        DC.W    $CDEF ; no code generated
```

Generated listing file

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1   000000 1234        Label:  DC.W    $1234
    2    2   000002 5678                DC.W    $5678
```

ENDFOR - End of FOR block

Syntax:

ENDFOR

Synonym:

None

Description

The ENDFOR directive indicates the end of a FOR block.

Note: The FOR directive is only available when the assembly option -Compat=b is used. By default, the FOR directive is not supported.

Example

see example of directive FOR.

See also

Directive FOR

Option -Compat

ENDIF - End Conditional Assembly

Syntax:

```
ENDIF
```

Synonym:

```
ENDC
```

Description

The `ENDIF` directive indicates the end of a conditional block. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

Example

see example of directive `IF`.

ENDM - End Macro Definition

Syntax:

```
ENDM
```

Synonym:

```
None
```

Description

The `ENDM` directive terminates the macro definition.

Example

```
cpChar:  MACRO
           LDAA \1
           STAA \2
         ENDM
DataSec: SECTION
char1:   DS 1
char2:   DS 1
CodeSec: SECTION
Start:
         cpChar char1, char2
```

EQU - Equate Symbol Value

Syntax:

```
<label>: EQU <expression>
```

Synonym:

None

Description

The EQU directive assigns the value of the <expression> in the operand field to <label>. The <label> and <expression> fields are both required, and the <label> cannot be defined anywhere else in the program. The <expression> cannot include a symbol, which is undefined or not defined yet.

The EQU directive does not allow forward references.

Example

```
         0000 0014  MaxElement: EQU  20
         0000 0050  MaxSize:    EQU  MaxElement * 4

  000000            Time:   DS.B 3
         0000 0000  Hour:   EQU Time  ; first byte addr.
         0000 0002  Minute: EQU Time+1; second byte addr
         0000 0004  Second: EQU Time+2; third byte addr
```

EVEN - Force Word Alignment

Syntax:

EVEN

Synonym:

None

Description

This directive forces the next instruction to the next even address relative to the start of the section. EVEN is an abbreviation for ALIGN 2. Some processors require word and long word operations to begin at even address boundaries. In such cases, the use of the EVEN directive ensures correct alignment, omission of the directive can result in an error message.

Example

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1   000000                   ds.b  4
   2    2                      ; location count has an even value
   3    3                      ; no padding byte inserted.
   4    4                            even
   5    5   000004                   ds.b  1
   6    6                      ; location count has an odd value
   7    7                      ; one padding byte inserted.
   8    8   000005                   even
   9    9   000006                   ds.b  3
  10   10                      ; location count has an odd value
  11   11                      ; one padding byte inserted.
  12   12   000009                   even
  13   13          0000 000A   aaa:  equ   10
```

FAIL - Generate Error Message

Syntax:

```
FAIL <arg> | <string>
```

Synonym:

None

Description

The `FAIL` directive comes in three flavors, depending on the operand specified:

- If <arg> is a number in the range [0– 499], the assembler generates an error message, including the line number and argument of the directive. The assembler does not generate an object file.
- If `<arg>` is a number in the range [500–$FFFFFFFF], the assembler generates a warning message, including the line number and argument of the directive.
- If a string is supplied as operand, the assembler generates an error message, including the line number and the `<string>`. The assembler does not generate any object file.

The `FAIL` directive is primarily intended for use with conditional assembly, to detect user defined errors or warning conditions.

Example:

The following portion of code:

```
cpChar: MACRO
          IFC "\1", ""
            FAIL 200
            MEXIT
          ELSE
            LDAA \1
          ENDIF

          IFC "\2", ""
            FAIL 600
          ELSE
            STAA \2
          ENDIF
        ENDM
```

```
codSec: SECTION
Start:
        cpChar char1
```

Generates the following error message:

```
>> in "C:\metrowerks\demo\warnfail.asm", line 13, col 19, pos 226

          IFC "\2", ""
            FAIL 600
                  ^
WARNING A2332: FAIL found
Macro Call :            FAIL 600
```

The following portion of code:

```
cpChar: MACRO
          IFC "\1", ""
            FAIL 200
            MEXIT
          ELSE
            LDAA \1
          ENDIF

          IFC "\2", ""
            FAIL 600
          ELSE
            STAA \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
        cpChar , char2
```

Generates the following error message:

```
>> in "C:\metrowerks\demo\errfail.asm", line 6, col 19, pos 96

          IFC "\1", ""
            FAIL 200
                  ^
ERROR A2329: FAIL found
Macro Call :            FAIL 200
```

The following portion of code:

```
cpChar: MACRO
          IFC "\1", ""
```

```
            FAIL "A character must be specified as first parameter"
            MEXIT
          ELSE
            LDAA \1
          ENDIF

          IFC "\2", ""
            FAIL 600
          ELSE
            STAA \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
        cpChar , char2
```

Generates the following error message:

```
>> in "C:\metrowerks\demo\failmes.asm", line 7, col 17, pos 110

          IFC "\1", ""
            FAIL "A character must be specified as first parameter"
                  ^
ERROR A2338: A character must be specified as first parameter
Macro Call :   FAIL "A character must be specified as first parameter"
```

FOR - Repeat assembly block

Syntax:

```
FAIL <arg> | <string>
```

Synonym:

None

Description

The FOR directive is an inline macro, since it can generate multiple lines of assembly code from only one line of input code.

FOR takes an absolute expression and assembles the portion of code following it, the number of time represented by the expression. The FOR expression may be either a constant or a label previously defined using EQU or SET.

Note: The FOR directive is only available when the assembly option -Compat=b is used. By default, the FOR directive is not supported.

Example:

```
FOR label=2 TO 6
  DC.B label*7
ENDFOR
```

Following code is generated by the above source:

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                         FOR label=2 TO 6
   2    2                           DC.B label*7
   3    3                         ENDFOR
   4    2   000000 0E               DC.B label*7
   5    3                         ENDFOR
   6    2   000001 15               DC.B label*7
   7    3                         ENDFOR
   8    2   000002 1C               DC.B label*7
   9    3                         ENDFOR
  10    2   000003 23               DC.B label*7
  11    3                         ENDFOR
  12    2   000004 2A               DC.B label*7
  13    3                         ENDFOR
```

See also

Directive ENDFOR

Option -Compat

IF - Conditional Assembly

Syntax:

```
IF <condition>
 [<assembly language statements>]
[ELSE]
 [<assembly language statements>]
ENDIF
```

Synonym:

None

Description

If `<condition>` is true, the statements immediately following the `IF` directive are assembled. Assembly continues until the corresponding `ELSE` or `ENDIF` directive is reached. Then all the statements until the corresponding `ENDIF` directive are ignored. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

The expected syntax for `<condition>` is:

```
<condition> := <expression> <relation> <expression>
<relation>  :=  "=" | "!=" | " >=" | ">" | "<=" | "<" | "<>"
The <expression> must be absolute (It must be known at assembly
time).
```

Example

The following is an example of the use of conditional assembly directives:

```
Try: EQU 0
     IF  Try != 0
       LDAA   #103
     ELSE
       LDAA   #0
     ENDIF
```

The value of *Try* determines the instruction to be assembled in the program. As shown, the "*ldaa #0"* instruction is assembled. Changing the operand of the "*equ"* directive to one causes the "*ldaa #103"* instruction to be assembled instead. The

following shows the listing provided by the assembler for these lines of code:

```
1    1           0000 0000   Try: EQU 0
2    2           0000 0000        IF  Try != 0
4    4                            ELSE
4    4   000000 8667                  LDAA #103
6    6                            ENDIF
```

IFcc - Conditional Assembly

Syntax:

```
IFcc <condition>
  [<assembly language statements>]
[ELSE]
  [<assembly language statements>]
ENDIF
```

Synonym:

None

Description

These directives can be replaced by the `IF` directive `Ifcc <condition>` is true, the statements immediately following the `Ifcc` directive are assembled. Assembly continues until the corresponding `ELSE` or `ENDIF` directive is reached, after which assembly moves to the statements following the `ENDIF` directive. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

The following table lists the available conditional types:

Ifcc	Condition	Meaning
ifeq	<expression>	if <expression> == 0
ifne	<expression>	if <expression> != 0
iflt	<expression>	if <expression> < 0
ifle	<expression>	if <expression> <= 0
ifgt	<expression>	if <expression> > 0
ifge	<expression>	if <expression> >= 0
ifc	<string1>, <string2>	if <string1> == <string2>
ifnc	<string1>, <string2>	if <string1> != <string2>
ifdef	<label>	if <label> was defined
ifndef	<label>	if <label> was not defined

Example

The following is an example of the use of conditional assembly directives:

```
Try: EQU 0
     IFNE  Try
       LDAA #103
      ELSE
       LDAA #0
      ENDIF
```

The value of *Try* determines the instruction to be assembled in the program. As shown, the "*ldaa #0"* instruction is assembled. Changing the directive to "*IFEQ"* causes the "*ldaa #103"* instruction to be assembled instead. The following shows the listing provided by the assembler for these lines of code:

```
1    1          0000 0000   Try: EQU 0
2    2          0000 0000        IFNE  Try
4    4                           ELSE
5    5   000000 8600              LDAA #0
6    6                           ENDIF
```

INCLUDE - Include Text from Another File

Syntax:

```
INCLUDE <file specification>
```

Synonym:

None

Description

This directive causes the included file to be inserted in the source input stream. The `<file specification>` is not case sensitive, and must be enclosed in quotation marks.

The assembler attempts to open `<file specification>` relative to the current working directory. If the file is not found there, then it is searched for relative to each path specified in the environment variable `GENPATH`.

Example

```
INCLUDE "..\LIBRARY\macros.inc"
```

LIST - Enable Listing

Syntax

```
LIST
```

Synonym:

None

Description

Specifies that the following instructions must be inserted in the listing and in the debug file. This option is selected by default. The listing file is only generated if the option -L is specified on the command line.

The source text following the `LIST` directive is listed until a `NOLIST` or an `END` is reached

This directive is not written to the listing and debug file.

Example:

The following portion of code:

```
aaa:    NOP

        LIST
bbb:    NOP
        NOP

        NOLIST
ccc:    NOP
        NOP

        LIST
ddd:    NOP
        NOP
```

generates the following listing file:

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
    1    1   000000 A7          aaa:    NOP
    2    2
    4    4   000001 A7          bbb:    NOP
```

```
    5    5   000002 A7                NOP
    6    6
   12   12   000005 A7          ddd:  NOP
   13   13   000006 A7                NOP
```

See Also

NOLIST

LLEN - Set Line Length

Syntax:

`LLEN <n>`

Synonym:

None

Description

Sets the number of characters from the source line that are included on the listing line to `<n>`. The values allowed for `<n>` are in the range `[0 – 132]`. If a value smaller than `0` is specified, the line length is set to `0`. If a value bigger than `132` is specified, the line length is set to `132`.

Lines of the source file that exceed the specified number of characters are truncated in the listing file.

Example:

The following portion of code:

```
        DC.B    $55

        LLEN    32
        DC.W    $1234, $4567

        LLEN    24
        DC.W    $1234, $4567
        EVEN
```

generates the following listing file:

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1   000000 55                  DC.B    $55
   2    2
   4    4   000001 1234 4567           DC.W    $1234, $4567
   5    5
   7    7   000005 1234 4567           DC.W    $1234, $
   8    8   000009 00                  EVEN
```

LONGEVEN - Forcing Long-Word Alignment

Syntax:

LONGEVEN

Synonym:

None

Description

This directive forces the next instruction to the next long-word address relative to the start of the section. LONGEVEN is an abbreviation for ALIGN 4.

Example

```
 2    2   000000 01                    dcb.b 1,1
           ; location counter is not a multiple of 4, 3 filling
           ; bytes are required.
 3    3   000001 0000 00               longeven
 4    4   000004 0002 0002             dcb.w 2,2
          ; location counter is already a multiple of 4, no filling
           ; bytes are required.
 5    5                                longeven
 6    6   000008 0202                  dcb.b 2,2
 7    7    ; following is for text section
 8    8                     s27        SECTION 27
 9    9   000000 9D                    nop
           ; location counter is not a multiple of 4, 3 filling
           ; bytes are required.
10   10   000001 0000 00               longeven
11   11   000004 9D                    nop
```

MACRO - Begin Macro Definition

Syntax:

```
<label>: MACRO
```

Synonym:

```
None
```

Description

The `<label>` of the `MACRO` directive is the name by which the macro is called. This name must not be a processor machine instruction or assembler directive name. For more information on macros, see the *Macros* chapter.

Example

```
          XDEF Start
MyData:   SECTION
char1:    DS.B  1
char2:    DS.B  1
cpChar:   MACRO
            LDAA \1
            STAA \2
          ENDM
CodeSec:  SECTION
Start:
          cpChar char1, char2
```

MEXIT - Terminate Macro Expansion

Syntax:

`MEXIT`

Synonym:

None

Description

`MEXIT` is usually used together with conditional assembly within a macro. In that case it may happen that the macro expansion should terminate prior to termination of the macro definition. The `MEXIT` directive causes macro expansion to skip any remaining source lines ahead of the `ENDM` directive.

Example

The following portion of code:

```
            XDEF  entry

storage:    EQU $00FF

save:       MACRO  ; Start macro definition
             LDX  #storage
             LDAA  \1
             STAA  0,x  ;save first arg
             LDAA  \2
             STAA  2,x  ;save second arg
             IFC  '\3', '';is there a 3rd arg?
               MEXIT    ; no, exit from macro.
             ENDC
             LDAA  \3   ; save third arg
             STAA  4,X
            ENDM     ; End of macro definition

datSec:     SECTION
char1:      ds.b 1
char2:      ds.b 1

codSec:     SECTION
entry:
            save char1, char2
```

generates the following listing file:

```
HC12-Assembler

 Abs.Rel.   Loc    Obj. code   Source line
 --------   ------ ---------   -----------
  1   1                              XDEF  entry
  2   2
  3   3          0000 00FF   storage:  EQU $00FF
  4   4
  5   5                      save:     MACRO  ; Start macro definition
  6   6                                  LDX  #storage
  7   7                                  LDAA  \1
  8   8                                  STAA  0,x  ;save first arg
  9   9                                  LDAA  \2
 10  10                                  STAA  2,x  ;save second arg
 11  11                                IFC  '\3', '';is there a 3rd arg?
 12  12                                  MEXIT    ; no, exit from macro.
 13  13                                  ENDC
 14  14                                  LDAA  \3   ; save third arg
 15  15                                  STAA  4,X
 16  16                                ENDM     ; End of macro definition
 17  17
 18  18                       datSec:   SECTION
 19  19   000000              char1:    ds.b 1
 20  20   000001              char2:    ds.b 1
 21  21
 22  22                       codSec:   SECTION
 23  23                       entry:
 24  24                                  save char1, char2
 25   6m  000000 CE 00FF    +            LDX  #storage
 26   7m  000003 B6 xxxx    +            LDAA  char1
 27   8m  000006 6A00       +            STAA  0,x    ; save first arg
 28   9m  000008 B6 xxxx    +            LDAA  char2
 29  10m  00000B 6A02       +            STAA  2,x   ; save second arg
 30  11m        0000 0001  +           IFC  '', '' ;is there a 3rd arg?
 32  12m                   +           MEXIT      ;no, exit from macro.
 33  13m                    +            ENDC
 34  14m                   +           LDAA       ; save third argument
 35  15m                    +            STAA  4,X
```

MLIST - List Macro Expansions

Syntax:

```
MLIST [ON | OFF]
```

Description

When the `ON` keyword is entered with an `MLIST` directive, the assembler includes the macro expansions in the listing and in the debug file.

When the `OFF` keyword is entered, the macro expansions are omitted from the listing and from the debug file.

This directive is not written to the listing and debug file, and the default value is `ON`.

Synonym:

None

Example

For the following code, with `MLIST ON`,

```
         XDEF  entry
         MLIST ON
swap:    MACRO
           LDD   \1
           LDX   \2
           STD   \2
           STX   \1
         ENDM
codSec: SECTION
entry:
         LDD   #$F0
         LDX   #$0F
main:
         STD   first
         STX   second
         swap  first, second
         NOP
         BRA   main
datSec: SECTION
first:  DS.W  1
second: DS.W  1
```

the assembler listing file is:

```
HC12-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      XDEF  entry
    3    3                      swap:   MACRO
    4    4                                LDD   \1
    5    5                                LDX   \2
    6    6                                STD   \2
    7    7                                STX   \1
    8    8                      ENDM
    9    9                      codSec: SECTION
   10   10                      entry:
   11   11   000000 CC 00F0             LDD   #$F0
   12   12   000003 CE 000F             LDX   #$0F
   13   13                      main:
   14   14   000006 7C xxxx             STD   first
   15   15   000009 7E xxxx             STX   second
   16   16                             swap  first, second
   17    4m  00000C FC xxxx    +          LDD   first
   18    5m  00000F FE xxxx    +          LDX   second
   19    6m  000012 7C xxxx    +          STD   second
   20    7m  000015 7E xxxx    +          STX   first
   21   17   000018 A7                   NOP
   22   18   000019 20EB                 BRA   main
   23   19                      datSec: SECTION
   24   20   000000             first:  DS.W  1
   25   21   000002             second: DS.W  1
```

For the same code, with `MLIST OFF`, the listing file is:

```
HC12-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      XDEF  entry
    3    3                      swap:   MACRO
    4    4                                LDD   \1
    5    5                                LDX   \2
    6    6                                STD   \2
    7    7                                STX   \1
    8    8                      ENDM
    9    9                      codSec: SECTION
   10   10                      entry:
   11   11   000000 CC 00F0             LDD   #$F0
   12   12   000003 CE 000F             LDX   #$0F
   13   13                      main:
```

```
14   14   000006 7C xxxx            STD   first
15   15   000009 7E xxxx            STX   second
16   16                             swap  first, second
21   17   000018 A7                 NOP
22   18   000019 20EB               BRA   main
23   19                   datSec:   SECTION
24   20   000000          first:    DS.W  1
25   21   000002          second:   DS.W  1
```

The `MLIST` directive does not appear in the listing file. When a macro is called after a `MLIST ON`, it is expanded in the listing file. If the `MLIST OFF` is encountered before the macro call, the macro is not expanded in the listing file.

NOLIST - Disable Listing

Syntax:

```
NOLIST
```

Synonym:

```
NOL
```

Description

Suppresses the printing of the following instructions in the assembly listing and debug file until a `LIST` directive is reached.

Example

The following portion of code:

```
aaa:    NOP

        LIST
bbb:    NOP
        NOP

        NOLIST
ccc:    NOP
        NOP

        LIST
ddd:    NOP
        NOP
```

generates the following listing file:

```
HC12-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1   000000 A7          aaa:    NOP
    2    2
    4    4   000001 A7          bbb:    NOP
    5    5   000002 A7                  NOP
    6    6
   12   12   000005 A7          ddd:    NOP
   13   13   000006 A7                  NOP
```

See Also

LIST Directive

NOPAGE - Disable Paging

Syntax:

NOPAGE

Synonym:

None

Description

Disables pagination in the listing file. Program lines are listed continuously, without headings or top or bottom margins.

OFFSET - Create Absolute Symbols

Syntax:

```
OFFSET <expression>
```

Synonym:

None

Description

The `OFFSET` directive declares an offset section and initializes the location counter to the value specified in `<expression>`. The `<expression>` must be absolute and may not contain references to external, undefined or forward defined labels.

An offset section is useful to simulate data structures or a stack frame.

Example:

The following example shows how you can use the OFFSET directive to access elements of a structure.

```
          OFFSET 0
ID:       DS.B   1
COUNT:    DS.W   1
VALUE:    DS.L   1
SIZE:     EQU *

DataSec: SECTION
Struct:  DS.B SIZE

CodeSec: SECTION
entry:
          LDX  #Struct
          LDAA  #0
          STAA  ID, X
          INC  COUNT, X
          INCA
          STAA  VALUE, X
```

When a statement affecting the location counter other than EVEN, LONGEVEN, ALIGN or DS is encountered after the OFFSET directive, the offset section is ended up. The preceding section is activated again, and the location counter is restored to the next available location in this section.

Example:

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                              OFFSET 0
    2    2   000000              ID:      DS.B   1
    3    3   000001              COUNT:   DS.W   1
    4    4   000003              VALUE:   DS.L   1
    5    5          0000 0007    SIZE:    EQU *
    6    6
    7    7                       DataSec: SECTION
    8    8   000000              Struct:  DS.B SIZE
    9    9
   10   10                       CodeSec: SECTION
   11   11                       entry:
   12   12   000000 CExx xx               LDX   #Struct
   13   13   000003 8600                 LDAA  #0
   14   14   000005 6A00                 STAA  ID, X
   15   15   000007 6201                 INC   COUNT, X
   16   16   000009 42                   INCA
   17   17   00000A 6A03                 STAA  VALUE, X
```

In the example above, the symbol ‘cst3’, defined after the OFFSET directive, defines a constant byte value. This symbol is appended to the section ‘ConstSec’, which precedes the OFFSET directive.

ORG - Set Location Counter

Syntax:

```
ORG <expression>
```

Synonym:

```
None
```

Description

The `ORG` directive sets the location counter to the value specified by `<expression>`. Subsequent statements are assigned memory locations starting with the new location counter value. The `<expression>` must be absolute and may not contain any forward, undefined, or external references. The `ORG` directive generates an internal section, which is absolute (see the *Sections* chapter).

Example

```
        org   $2000
b1:     nop
b2:     rts
```

Label `b1` is located at address `$2000` and label `b2` at address `$2001`:

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                          org   $2000
   2    2  a002000 A7          b1:   nop
   3    3  a002001 3D          b2:   rts
```

See also

SECTION Directive

DC Directive

DCB Directive

DS Directive

PAGE - Insert Page Break

Syntax:

PAGE

Synonym:

None

Description

Insert a page break in the assembly listing.

Example

The following portion of code:

```
code:   SECTION
        DC.B    $00,$12
        DC.B    $00,$34
        PAGE
        DC.B    $00,$56
        DC.B    $00,$78
```

generates the following listing file:

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      code:   SECTION
   2    2   000000 0012                DC.B    $00,$12
   3    3   000002 0034                DC.B    $00,$34


Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   5    5   000004 0056                DC.B    $00,$56
   6    6   000006 0078                DC.B    $00,$78
```

PLEN - Set Page Length

Syntax:

```
PLEN <n>
```

Synonym:

```
None
```

Description

Sets the listings page length to `<n>` lines. `<n>` may range from `10` to `10000`. If the number of lines already listed on the current page is greater than or equal to `<n>`, listing will continue on the next page with the new page length setting.

The default page length is 65 lines.

RAD50 - Rad50 encoded string constants

Syntax:

```
RAD50 <str>[, cnt]
```

Synonym:

```
None
```

Description

This directive places strings encoded with the RAD50 encoding into constants. The RAD50 encoding does place 3 string characters out of a reduced character set into 2 bytes. It therefore saves memory when comparing it with a plain ASCII representation. It also has some drawbacks, however. The only 40 different character values are supported and the strings have to be decoded before they can be used. This decoding does include some computations including divisions (not just shifts) and is therefore rather expensive.

The encode takes three bytes, looks them up in a string table.

```
unsigned short LookUpPos(char x) {
  static const char translate[]=
     " ABCDEFGHIJKLMNOPQRSTUVWXYZ$.?0123456789";
  const char* pos= strchr(translate, x);
  if (pos == NULL) { EncodingError(); return 0; }
  return pos-translate;
}
unsigned short Encode(char a, char b, char c) {
  return LookUpPos(a)*40*40 + LookUpPos(b)*40 + LookUpPos(c);
}
```

If the remaining string is shorter than 3 bytes, it is filled with spaces (which correspond to the RAD50 character 0).

The optional argument cnt can be used to explicitly state how many 16 bit values should be written. If the string is shorter than 3*cnt, then it is filled with spaces.

See the example C code below how to decode it.

Example:

The data in the following file:

```
  XDEF rad50, rad50Len
DataSection  SECTION
rad50:
          RAD50 "Hello World"
rad50Len: EQU (*-rad50)/2
```

assembles to the following data:

```
    $32D4 $4D58 $922A $4BA0
```

This C code takes the data and actually prints “Hello World”:

```
#include "stdio.h"
extern unsigned short rad50[];
extern int rad50Len; /* address is value. Exported asm label */
#define rad50len ((int) &rad50Len)

void printRadChar(char ch) {
   static const char translate[]=
     " ABCDEFGHIJKLMNOPQRSTUVWXYZ$.?0123456789";
   char asciiChar= translate[ch];
   (void)putchar(asciiChar);
}
void PrintHallo(void) {
  unsigned char values= rad50len;
  unsigned char i;
  for (i=0; i < values; i++) {
    unsigned short val= rad50[i];
    printRadChar(val / (40 * 40));
    printRadChar((val / 40) % 40);
    printRadChar(val % 40);
  }
}
```

SECTION - Declare Relocatable Section

Syntax:

```
<name>:  SECTION [SHORT][<number>]
```

Synonym:

None

Description

This directive declares a relocatable section and initializes the location counter for the following code. The first `SECTION` directive for a section sets the location counter to zero. Subsequent `SECTION` directives for that section restore the location counter to the value that follows the address of the last code in the section.

`<name>` is the name assigned to the section. Two `SECTION` directives with the same name specified refer to the same section.

`<number>` is optional and is only specified for compatibility with MASM assembler.

A section is a code section when it contains at least one assembly instruction. It is considered to be a constant section if it contains only `DC` or `DCB` directives. A section is considered to be a data section when it contains at least a `DS` directive or if it is empty.

Example

The following example demonstrates the definition of a section `aaa`, which is splitted in two blocks, with section `bbb` in-between them.

The location counter associated with the label `zz` is 1, because a `NOP` instruction was already defined in this section at label `xx`.

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------

   1    1                       aaa:    SECTION 4
   2    2   000000 A7           xx:     NOP
   3    3                       bbb:    SECTION 5
   4    4   000000 A7           yy:     NOP
```

```
   5    5   000001 A7                NOP
   6    6   000002 A7                NOP
   7    7                    aaa:    SECTION 4
   8    8   000001 A7        zz:     NOP
```

The optional qualifier SHORT specifies that the section is a short section, That means than the objects defined there can be accessed using the direct addressing mode.

Example:

The following example demonstrates the definition and usage of a SHORT section.

In following example, the symbol data is accessed using the direct addressing mode.

```
HC12-Assembler
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      dataSec: SECTION  SHORT
    2    2   000000             data:    DS.B 1
    3    3
    4    4                      codeSec: SECTION
    5    5
    6    6                      entry:
    7    7   000000 87                   CLRA
    8    8   000001 5Axx                 STAA data
```

See also

ORG Directive

DC Directive

DCB Directive

DS Directive

SET - Set Symbol Value

Syntax:

```
<label>: SET <expression>
```

Synonym:

None

Description

Similar to the EQU directive, the SET directive assigns the value of the <expression> in the operand field to the symbol in the <label> field. The <expression> must resolve as an absolute expression and cannot include a symbol that is undefined or not yet defined. The <label> is an assembly time constant, SET does not generate any machine code.

The value is temporary; a subsequent SET directive can redefine it.

Example

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1          0000 0002   count:  SET     2
   2    2   000000 02          one:    DC.B    count
   3    3
   4    4          0000 0001   count:  SET     count-1
   5    5   000001 01                  DC.B    count
   6    6
   7    7          0000 0001           IFNE count
   8    8          0000 0000   count:  SET     count-1
   9    9                              ENDIF
  10   10   000002 00                  DC.B    count
```

The value associated with the label count is decremented after each DC.B instruction.

SPC - Insert Blank Lines

Syntax:

```
SPC <count>
```

Synonym:

```
None
```

Description

Inserts `<count>` blank lines in the assembly listing. `<count>` may range from 0 to 65. This has the same effect as writing that number of blank lines in the assembly source. A blank line is a line containing only a carriage return.

TABS - Set Tab Length

Syntax:

```
TABS <n>
```

Synonym:

```
None
```

Description

Sets the tab length to <n> spaces. The default tab length is eight. <n> may range from 0 to 128.

TITLE - Provide Listing Title

Syntax:

```
TITLE "title"
```

Synonym:

```
TTL
```

Description

Print the `<title>` on the head of every page of the listing file. This directive must be the first source code line. A title consists of a string of characters enclosed in quotes (").

The title specified will be written on the top of each page in the assembly listing file.

XDEF - External Symbol Definition

Syntax:

```
XDEF [.<size>] <label>[,<label>]...
```

where <size> = W(default)

Synonym:

```
GLOBAL, PUBLIC
```

Description

This directive specifies labels defined in the current module that are to be passed to the linker as labels that can be referenced by other modules linked to the current module.

The number of symbols enumerated in a `XDEF` directive is only limited by the memory available at assembly time.

Example

```
          XDEF Count, main
          ;; variable Count can be referenced in other modules,
          ;; same for label main. Note that linker and assembler
          ;; are case-sensitive, i.e., Count != count.

Count:    DS.W  2

code:     SECTION
main:     DC.B 1
```

XREF - External Symbol Reference

Syntax:

```
XREF [.<size>] <symbol>[,<symbol>]...
```

where <size> = W(default)

Synonym:

EXTERNAL

Description

This directive specifies symbols referenced in the current module but defined in another module. The list of symbols and corresponding 32 - bit values is passed to the linker.

The number of symbols enumerated in a XREF directive is only limited by the memory available at assembly time.

Example

```
XREF   OtherGlobal  ; Reference "OtherGlobal" defined in another
                    ; module (See XDEF directive example.)
```

XREFB - External Reference for Symbols located on the Direct Page

Syntax:

```
XREFB <symbol>[,<symbol>]...
```

Synonym:

None

Description:

This directive specifies symbols referenced in the current module but defined in another module. Symbols enumerated in a XREFB directive, can be accessed using the direct address mode. The list of symbols and corresponding 8-bit values is passed to the linker.

The number of symbols enumerated in a XREFB directive is only limited by the memory available at assembly time.

Example:

```
      XREFB   OtherDirect  ; Reference "OtherDirect" defined in another
                           ; module (See XDEF directive example.)
```

Macros

A macro is a template for a code sequence. Once a macro is defined, subsequent reference to the macro name are replaced by its code sequence.

Macro Overview

A macro must be defined before it is called. When a macro is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently called.

The assembler expands the macro definition each time the macro is called. The macro call causes source statements to be generated, which may include macro arguments. A macro definition may contain any code or directive except nested macro definitions. Calling previously defined macros is also allowed. Source statements generated by a macro call are inserted in the source file at the position where the macro is invoked.

To call a macro, write the macro name in the operation field of a source statement. Place the arguments in the operand field. The macro may contain conditional assembly directives that cause the assembler to produce in-line-coding variations of the macro definition.

Macros call produces in-line code to perform a predefined function. Each time the macro is called, code is inserted in the normal flow of the program so that the generated instructions are executed in line with the rest of the program.

Defining a Macro

The definition of a macro consists of four parts:

- The header statement, a `MACRO` directive with a label that names the macro.
- The body of the macro, a sequential list of assembler statements, some possibly including argument placeholders.
- The `ENDM` directive, terminating the macro definition.
- eventually an instruction `MEXIT`, which stops macro expansion.

See *Section Assembler Directives* for information about the `MACRO`, `ENDM`, `MEXIT`, `MLIST` directives.

The body of a macro is a sequence of assembler source statements. Macro parame-

ters are defined by the appearance of parameter designators within these source statements. Valid macro definition statements includes the set of processor assembly language instructions, assembler directives, and calls to previously defined macros. However, macro definitions may not be nested.

Calling Macros

The form of a macro call is:

```
[<label>:] <name>[.<sizearg>] [<argument> [,<argument>]...]
```

Although a macro may be referenced by another macro prior to its definition in the source module, a macro must be defined before its first call. The name of the called macro must appear in the operation field of the source statement. Arguments are supplied in the operand field of the source statement, separated by commas.

The macro call produces in-line code at the location of the call, according to the macro definition and the arguments specified in the macro call. The source statements of the expanded macro are then assembled subject to the same conditions and restrictions affecting any source statement. Nested macros calls are also expanded at this time.

Macro Parameters

As many as 36 different substitutable parameters can be used in the source statements that constitute the body of a macro. These parameters are replaced by the corresponding arguments in a subsequent call to that macro.

A parameter designator consists of a backlashes character (\), followed by a digit (0 - 9) or an uppercase letter (A - Z). Parameter designator \0 corresponds to a size argument that follows the macro name, separated by a period (.).

Example

Consider the following macro definition:

```
MyMacro: MACRO
           DC.\0   \1, \2
         ENDM
```

When this macro is used in a program, e.g.:

```
         MyMacro.B $10, $56
```

the assembler expands it to:

```
DC.B $10, $56
```

Arguments in the operand field of the macro call refer to parameter designator \1 through \9 and \A through \Z, in that order. The argument list (operand field) of a macro call cannot be extended onto additional lines.

At the time of a macro call, arguments from the macro call are substituted for parameter designators in the body of the macro as literal (string) substitutions. The string corresponding to a given argument is substituted literally wherever that parameter designator occurs in a source statement as the macro is expanded. Each statement generated in the execution is assembled in line.

It is possible to specify a null argument in a macro call by a comma with no character (not even a space) between the comma and the preceding macro name or comma that follows an argument. When a null argument itself is passed as an argument in a nested macro call, a null value is passed. All arguments have a default value of null at the time of a macro call.

Macro Argument Grouping

To pass text including commas as a single macro argument, the assembler supports a special syntax. This grouping starts with the [? prefix and ends with the ?] suffix. If the [? or ?] patterns occur inside of the argument text, they have to be in pairs. Alternatively, brackets, question marks and backward slashes can also be escaped with a backward slash as prefix.

Note: This escaping only takes place inside of [? ?] arguments. A backward slash is only removed in this process if it is just before a bracket ([]), a question mark (?) or a second backwards slash (\).

Example

```
MyMacro:  MACRO
            DC   \1
          ENDM
MyMacro1: MACRO
            \1
          ENDM
```

Here some macro calls with rather complicated arguments:

```
MyMacro [?$10, $56?]
MyMacro [?"\[?"?]
MyMacro1 [?MyMacro [?$10, $56?]?]
MyMacro1 [?MyMacro \[?$10, $56\?]?]
```

These macro calls expand to the following lines:

```
        DC   $10, $56
        DC   "[?"
        DC   $10, $56
        DC   $10, $56
```

The macro assembler does also supports for compatibility with previous versions macro grouping with a angle bracket syntax:

```
MyMacro <$10, $56>
```

However, this old syntax is ambiguous as < and > are also used as compare operators. For example the following code does not produce the expected result:

```
MyMacro <1 > 2, 2 > 3> ; Wrong!
```

Because of this the old angle brace syntax should be avoided in new code. There is also and option to disable it explicitly.

See also the *option -CMacBrackets* and the *option -CMacAngBrack*.

Labels Inside Macros

To avoid the problem of multiple-defined labels resulting from multiple calls to a macro that has labels in its source statements, the programmer can direct the assembler to generate unique labels on each call to a macro.

Assembler-generated labels include a string of the form _nnnnn where nnnnn is a 5 digit value. The programmer requests an assembler-generated label by specifying \@ in a label field within a macro body. Each successive label definition that specifies a \@ directive generates a successive value of _nnnnn, thereby creating a unique label on each macro call. Note that \@ may be preceded or followed by additional characters for clarity and to prevent ambiguity.

Example

This is the definition of the clear macro:

```
clear:   MACRO
           LDX    #\1
           LDAA   #16
\@LOOP:    CLR    1,X+
           DBNE   A,\@LOOP
         ENDM
```

This macro is called in the application:

```
Data: Section
temporary: DS 16
```

```
data:      DS 16

Code: Section
         clear     temporary
         clear     data
```

The two macro calls of `clear` are expanded in the following manner:

```
HC12-Assembler
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      clear:   MACRO
    2    2                                 LDX    #\1
    3    3                                 LDAA   #16
    4    4                      \@LOOP:    CLR    1,X+
    5    5                                 DBNE   A,\@LOOP
    6    6                               ENDM
    7    7
    8    8                      Data: Section
    9    9   000000             temporary: DS 16
   10   10   000010             data:      DS 16
   11   11
   12   12                      Code: Section
   13   13                               clear     temporary
   14    2m  000000 CE xxxx    +           LDX    #temporary
   15    3m  000003 8610       +           LDAA   #16
   16    4m  000005 6930       +_00001LOOP:    CLR    1,X+
   17    5m  000007 0430 FB    +           DBNE   A,_00001LOOP
   18   14                               clear     data
   19    2m  00000A CE xxxx    +           LDX    #data
   20    3m  00000D 8610       +           LDAA   #16
   21    4m  00000F 6930       +_00002LOOP:    CLR    1,X+
   22    5m  000011 0430 FB    +           DBNE   A,_00002LOOP
```

Macro Expansion

When the assembler reads a statement in a source program calling a previously defined macro, it processes the call as described in the following paragraphs.

The symbol table is searched for the macro name. If it is not in the symbol table, an undefined symbol error message is issued.

The rest of the line is scanned for arguments. Any argument in the macro call is saved as a literal or null value in one of the 35 possible parameter fields. When the number of arguments in the call is less than the number of parameters used in the macro the argument, which have not been defined at invocation time are initialize with “” (empty string).

Starting with the line following the MACRO directive, each line of the macro body is saved and is associated with the named macro. Each line is retrieved in turn, with parameter designators replaced by argument strings or assembler-generated label strings.

Once the macro is expanded, the source lines are evaluated and object code is produced.

Nested Macros

Macro expansion is performed at invocation time, which is also the case for nested macros. If the macro definition contains nested macro call, the nested macro expansion takes place in line. Recursive macro call are also supported.

A macro call is limited to the length of one line, i.e. 1024 characters.

Assembler Listing File

The assembly listing file is the output file of the assembler, which contains information about the generated code. The listing file is generated when the option –L is activated. When an error is detected during assembling from the file, there is no listing file generated.

The amount of information available depends on following assembly options:

Option -L
Option -Lc
Option -Ld
Option -Le
Option -Li

The information in the listing file also depends on following assembly directives:

LIST, NOLIST, CLIST, MLIST.

The format from the listing file is influenced by following directives:

PLEN, LLEN, TABS, SPC, PAGE, NOPAGE, TITLE.

The name of the listing file generated is <base name>.lst

Page Header

The page header consist on 3 lines:

- The first line contains an optional user string defined in the directive TITLE.
- The second line contains the name of the assembler vendor (METROWERKS) as well as the target processor name (HC12).
- The third line contains a copyright notice.

Example

```
Demo Application
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1991-2001
```

Source Listing

The printed columns can be configured with the option -Lasmc. By default the following 5 columns are contained:

Abs.

This column contains the absolute line number for each instruction. The absolute line number is the line number in the debug listing file, which contains all included files and where all macro calls have been expanded.

Example

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      ;-------------------------------
   2    2                      ; File: test.o
   3    3                      ;-------------------------------
   4    4
   5    5                              XDEF Start
   6    6                      MyData:  SECTION
   7    7   000000             char1:   DS.B  1
   8    8   000001             char2:   DS.B  1
   9    9                              INCLUDE "macro.inc"
  10   1i                      cpChar:  MACRO
  11   2i                                LDAA \1
  12   3i                                STAA \2
  13   4i                              ENDM
  14   10                      CodeSec: SECTION
  15   11                      Start:
  16   12                              cpChar char1, char2
  17   2m  000000 B6 xxxx    +           LDAA char1
  18   3m  000003 7A xxxx    +           STAA char2
  19   13  000006 A7                     NOP
  20   14  000007 A7                     NOP
```

In the previous example, the line number displayed in the column ‘Abs.’ are incremented for each line.

Rel.

This column contains the relative line number for each instruction. The relative line number is the line number in the source file. For included files, the relative line number is the line number in the included file. For macro call expansion, the relative line number is the line number of the instruction in the macro definition.

A ‘i’ suffix is appended to the relative line number, when the line comes from an included file. A ‘m’ suffix is appended to the relative line number, when the line is generated by a macro call.

Example

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      ;--------------------------------
   2    2                      ; File: test.o
   3    3                      ;--------------------------------
   4    4
   5    5                              XDEF Start
   6    6                      MyData:  SECTION
   7    7   000000             char1:   DS.B  1
   8    8   000001             char2:   DS.B  1
   9    9                              INCLUDE "macro.inc"
  10   1i                      cpChar:  MACRO
  11   2i                               LDAA \1
  12   3i                               STAA \2
  13   4i                              ENDM
  14   10                      CodeSec: SECTION
  15   11                      Start:
  16   12                              cpChar char1, char2
  17   2m  000000 B6 xxxx      +          LDAA char1
  18   3m  000003 7A xxxx      +          STAA char2
  19   13  000006 A7                      NOP
  20   14  000007 A7                      NOP
```

In the previous example, the line number displayed in the column ‘Rel.’ represent the line number of the corresponding instruction in the source file.

‘1i’ on absolute line number 10 denotes that the instruction ‘`cpChar: MACRO`’ is located in an included file.

‘2m’ on absolute line number 17 denotes that the instruction ‘`LDAA char1`’ is generated by a macro expansion.

Loc

This column contains the address of the instruction. For absolute sections, the address is preceded by a ‘a’ and contains the absolute address of the instruction. For relocatable sections, this address is the offset of the instruction from the beginning of the relocatable section.. This offset is a hexadecimal number coded on 6 digits.

A value is written in this column in front of each instruction generating code or allocating storage. This column is empty in front of each instruction which does not generate code (for example SECTION, XDEF, …).

Example

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      ;--------------------------------
   2    2                      ; File: test.o
   3    3                      ;--------------------------------
   4    4
   5    5                              XDEF Start
   6    6                      MyData:  SECTION
   7    7   000000             char1:   DS.B  1
   8    8   000001             char2:   DS.B  1
   9    9                              INCLUDE "macro.inc"
  10    1i                     cpChar:  MACRO
  11    2i                               LDAA \1
  12    3i                               STAA \2
  13    4i                             ENDM
  14   10                      CodeSec: SECTION
  15   11                      Start:
  16   12                              cpChar char1, char2
  17    2m  000000 B6 xxxx    +           LDAA char1
  18    3m  000003 7A xxxx    +           STAA char2
  19   13   000006 A7                    NOP
  20   14   000007 A7                    NOP
```

In the previous example, the hexadecimal number displayed in the column ‘Loc.’ is the offset of each instruction in the section ‘codeSec’.

There is no location counter specified in front of the instruction ‘`INCLUDE "macro.inc"`’ because this instruction does not generate code.

The instruction ‘`LDAA char1`’ is located at offset 0 from the section ‘codeSec’ start address.

The instruction ‘`STAA char2`’ is located at offset 3 from the section ‘codeSec’ start address.

Obj. Code

This column contains the hexadecimal code of each instruction in hexadecimal format. This code is not identical to the code stored in the object file. The letter ‘x’ is displayed at the position where the address of an external or relocatable label is expected. Code at position when ‘x’ are written will be determined at link time.

Example

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ----------- -----------
   1    1                      ;--------------------------------
   2    2                      ; File: test.o
   3    3                      ;--------------------------------
   4    4
   5    5                              XDEF Start
   6    6                      MyData:  SECTION
   7    7   000000             char1:   DS.B  1
   8    8   000001             char2:   DS.B  1
   9    9                              INCLUDE "macro.inc"
  10    1i                     cpChar:  MACRO
  11    2i                              LDAA \1
  12    3i                              STAA \2
  13    4i                              ENDM
  14   10                      CodeSec: SECTION
  15   11                      Start:
  16   12                              cpChar char1, char2
  17    2m  000000 B6 xxxx    +           LDAA char1
  18    3m  000003 7A xxxx    +           STAA char2
  19   13   000006 A7                    NOP
  20   14   000007 A7                    NOP
```

Source Line

This column contains the source statement. This is a copy of the source line from the source module. For lines resulting from a macro expansion, the source line is the expanded line, where parameter substitution has been done.

Example

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      ;--------------------------------
   2    2                      ; File: test.o
   3    3                      ;--------------------------------
   4    4
   5    5                              XDEF Start
   6    6                      MyData:  SECTION
   7    7   000000             char1:   DS.B  1
   8    8   000001             char2:   DS.B  1
   9    9                              INCLUDE "macro.inc"
  10    1i                     cpChar:  MACRO
  11    2i                              LDAA \1
  12    3i                              STAA \2
```

```
13    4i                                ENDM
14   10                      CodeSec: SECTION
15   11                      Start:
16   12                                cpChar char1, char2
17    2m  000000 B6 xxxx    +             LDAA char1
18    3m  000003 7A xxxx    +             STAA char2
19   13   000006 A7                     NOP
20   14   000007 A7                     NOP
```

MASM Compatibility

The macro assembler has been extended to ensure compatibility with the MASM assembler.

Comment Line

A line starting with a '*' character is considered to be a comment line by the assembler.

Constants

Integer Constants

For compatibility with the MASM assembler, following notations are also supported for integer constants:

- A decimal constant is defined by a sequence of decimal digits (0-9) followed by a 'd' or 'D' character.
- A hexadecimal constant is defined by a sequence of hexadecimal digits (0-9, a-f, A-F) followed by a 'h' or 'H' character.
- An octal constant is defined by a sequence of octal digits (0-7) followed by a 'o', 'O', 'q', or 'Q' character.
- A binary constant is defined by a sequence of binary digits (0-1) followed by a 'b' or 'B' character.

Example

```
512d        ; decimal representation
512D        ; decimal representation
200h        ; hexadecimal representation
200H        ; hexadecimal representation
1000o       ; octal representation
1000O       ; octal representation
1000q       ; octal representation
1000Q       ; octal representation
1000000000b ; binary representation
1000000000B ; binary representation
```

Operators

For compatibility with the MASM assembler, following notations are also supported for operators:

Operator	Notation
Shift left	!<
Shift right	!>
Bitwize AND	!.
Bitwize OR	!+
Bitwize XOR	!x, !X

Directives

The following table enumerates the directives, which are supported by the macro assembler for compatibility with MASM:

Operator	Notation	Description
RMB	DS	Define storage for a variable. Argument specifies the byte size
RMD	DS 2*	Define storage for a variable. Argument specifies the number of 2 byte blocks
RMQ	DS 4*	Define storage for a variable Argument specifies the number of 4 byte blocks
ELSEC	ELSE	Alternate of conditional block
ENDC	ENDIF	End of conditional block
NOL	NOLIST	Specify that all subsequent instructions must not be inserted in the listing file.
TTL	TITLE	Define the user defined title for the assembler listing file.
GLOBAL	XDEF	Make a symbol public (Visible from outside)
PUBLIC	XDEF	Make a symbol public (Visible from outside)
EXTERNAL	XREF	Import reference to an external symbol.

Operator	Notation	Description
XREFB	XREF.B	Import reference to an external symbol located on the direct page.
SWITCH		Allows to switch to a section which have been defined previously.
ASCT		Creates a predefined section which name id ASCT.
BSCT		Creates a predefined section which name id BSCT. Variable defined in this section are accessed using the direct addressing mode.
CSCT		Creates a predefined section which name id CSCT.
DSCT		Creates a predefined section which name id DSCT.
IDSCT		Creates a predefined section which name id IDSCT.
IPSCT		Creates a predefined section which name id IPSCT.
PSCT		Creates a predefined section which name id PSCT.

MCUasm Compatibility

The macro assembler has been extended to ensure compatibility with the MCUasm assembler.

MCUasm compatibility mode can be activated, specifying the option -MCUasm.

Labels

When MCUasm compatibility mode is activated, labels must be followed by a colon, even when they start on column 1.

Example

When MCUasm compatibility mode is activated, following portion of code generate an error message, because the label ‘label’ is not followed by a colon.

```
label       DC.B 1
```

When MCUasm compatibility mode is not activated, the previous portion of code does not generate any error message.

SET Directive

When MCUasm compatibility mode is activated, relocatable expressions are also allowed in a SET directive.

Example

When MCUasm compatibility mode is activated, following portion of code does not generate any error message:

```
label: SET *
```

When MCUasm compatibility mode is not activated, the previous portion of code generates any error message, because SET label can only refer to absolute expressions.

Obsolete Directives

The following table enumerates the directives, which are not recognized any more,

when MCUasm compatibility mode is switched ON.:

Operator	Notation	Description
RMB	DS	Define storage for a variable
NOL	NOLIST	Specify that all subsequent instructions must not be inserted in the listing file.
TTL	TITLE	Define the user defined title for the assembler listing file.
GLOBAL	XDEF	Make a symbol public (Visible from outside)
PUBLIC	XDEF	Make a symbol public (Visible from outside)
EXTERNAL	XREF	Import reference to an external symbol.

Semi-Avocet Compatibility

The macro assembler has been extended to ensure compatibility with the Avocet assembler.

Avocet compatibility mode can be activated, specifying the option -C=SAvocet. The compatibility does not cover all Avocet specific features but only some of them.

Directives

Macro Parameters

Section Definition

Structured Assembly

Directives

The following table enumerates the directives, which are supported when the Avocet Assembler compatibility mode is activated.:

Directive	Notation	Description
DEFSEG		Segment definition (See section "Section Definition" below).

ELSEIF		Conditional directive, checking a specific condition.
		`IF ((label1 & label2) != 0)`
		`  LDD #label1`
		`ELSIF (label1 = 0)`
		`  LDD #label2`
		`ELSE`
		`  LDD #0`
		`ENDIF`

EXITM	MEXIT	Define an exit condition for a macro.
		`Copy  MACRO  source, dest`
		`  IFB "source"`
		`  EXITM`
		`  ENDIF`
		`  LDD  source`
		`  STD  dest`
		`  ENDM`

Directive	Notation	Description

IFB Param	IFC Param, “”	Test if a macro parameter is empty. The syntax is IFB "param".
		`Copy MACRO source, des`
		`IFB "source"`
		`LDD #0`
		`STD dest`
		`ELSE`
		`LDD source`
		`STD dest`
		`ENDIF`
		`ENDM`

IFNB Param	IFNC Param “”	Test if a macro parameter is not empty. The syntax is IFNB "param"
		`Copy MACRO source, dest`
		`IFNB "source"`
		`LDD source`
		`STD dest`
		`ELSE`
		`LDD #0`
		`STD dest`
		`ENDIF`
		`ENDM`

NOSM	MLIST OFF	Do not insert macro expansion in listing file.
SEG	SWITCH	Switch to a previously defined segment (See section “Section Definition” below).
SM	MLIST ON	Insert macro expansion in listing file.
SUBTITLE		Defines a subtitle for the input file. This subtitle is written to the listing file. `SUBTITLE title2: Main File`
TEQ	SET	Define a constant, which value may be modified in the source file

Section Definition

Section definition is performed using the directive DEFSEG. The correct syntax for a DEFSEG directive is:

```
DEFSEG <name> [START=<start address>] [<section qualifier>]
```

Where:

- name: is the name of the section.
- start address: is the start address for the section. This parameter is optional.
- section qualifier: is the qualifier which applies to the section. This parameter is optional and may take the value:

Qualifier	Meaning
PAGE0	for data section located on the direct page
DATA	for data section
CODE	for code section

Example:

```
DEFSEG myDataSection
DEFSEG D_ATC_TABLES START=$0EA0
DEFSEG myDirectData PAGE0
```

Note: Because of an incompatibility in the object file format, an absolute section implementation must entirely reside in a single assembly unit. You cannot split the code from an absolute section over several object files. An absolute section is a section associated with a start address.

Note: In order to split a section over different assembly units, you better define the section as relocatable (without START) and you can specify in the linker PRM file the address where you want to load the section.

Example:

In the assembly source file:

```
DEFSEG D_ATC_TABLES ;START=$0EA0
```

In the linker parameter file:

```
...
SECTION
  ...
  MY_TABLE = READ_WRITE 0x0EA0 TO 0x0EFF;
PLACEMENT
  ...
 D_ATC_TABLES INTO MY_TABLE:
...
```

The directive SEG is then used to activate the corresponding section in the assembly source file.

The name specified in a SEG directive was always previously specified in a DEFSEG directive.

Following syntax will be accepted for SEG:

```
SEG <name>
```

where

name: is the name of the section, which was previously defined in a DEFSEG directive.

Example:

```
SEG myDataSection
```

Macro Parameters

When Avocet Compatibility is switched ON, names can be associated with macro parameters. A macro definition can now look as follows:

```
Copy        MACRO   source, destination
               LDD   source
               STD   destination
            ENDM
```

Support for Structured Assembly

When the Avocet compatibility is switched ON, SWITCH or FOR construct are available in Macro assembler.

Switch Block

The SWITCH directive evaluates an expression and assembles the code following the particular CASE statement, which satisfies the switch expression. If no CASE statement corresponds to the value of the expression, the code following the DEFAULT (if present) is assembled.

ENDSW terminates the SWITCH directive.

The expression specified in a SWITCH directive must be an absolute expression.

Example:

```
xxx   equ 5
   …
   SWITCH xxx
   CASE 0
      LDD #1
   CASE 1
     LDD 2
   CASE 3
     LDD #6
   DEFAULT
     LDD #0
   ENDSW
```

Following set of instructions are generated by the above portion of code:

```
xxx   equ 5
   …
   LDD #0
```

FOR Block

In the Avocet compatibility mode, the FOR direct§ive is supported.

Example:

```
   FOR l=2 TO 6
     NOP
   ENDF
```

Following set of instructions are generated by the above portion of code:

```
     NOP
     NOP
     NOP
     NOP
     NOP
```

Mix C and Assembler Applications

When you intend to mix Assembly source file and ANSI C source files in a single application, following issues are important:

- Memory Models
- Parameter Passing Scheme
- Return Value Location
- Accessing assembly variables in an ANSI C source file
- Accessing ANSI C variables in an assembly source file
- Invoking an assembly function in an ANSI C source file
- Support for Structured Types

To build mixed C and Assembler applications, you have to know how the C - Compiler uses registers and calls procedures. The following sections will describe this for compatibility with the compiler. If you are working with another vendor ANSI C compiler, refer to your Compiler Manual to get the information about parameter passing rules.

Memory Models

The memory models are only important if you mix C and assembler code. In this case all sources must be compiled or assembled with the same memory model.

The assembler supports all memory models of the compiler. Depending on your hardware use the smallest memory model suitable for your programming needs.

The table below summarizes the different memory models. It shows when to use a particular memory model and which assembler switch to use.

Option	Memory Model	Local Data	Global Data	Suggested Use
–Ms	SMALL	SP rel	extended	Small applications which fit into the 64k address space or which do only have limited places where paged area is accessed.

Option	Memory Model	Local Data	Global Data	Suggested Use
–Mb	BANKED	SP rel	extended	Larger applications which code does not fit into the 64k address space. Data is limited to the 64 k address space. The code generated by the compiler is not much larger as in the SMALL memory model because the CPU supports the CALL instruction. Usually there is one additional byte per function call.
–Ml	LARGE	SP rel	far	Applications which data does not fit into 64k address space. The code generated by the compiler is significantly larger then in the other memory models.

Note: The default pointer size for the compiler is also affected by the memory model chosen.

Parameter Passing Scheme

When you are using the HC12 compiler, the parameter passing scheme is the following:

The Pascal calling convention is used for functions with a fixed number of parameters: The caller pushes the arguments from left to right. After the call, the caller removes the parameters from the stack again.

The C calling convention is used only for functions with a variable number of parameters. In this case the caller pushes the arguments from right to left.

If the last parameter of a function with a fixed number of arguments has a simple type, it is not pushed but passed in a register. This results in shorter code because pushing the last parameter can be saved. The following table shows an overview of the registers used for argument passing.

Size of Last Parameter	Type example	Register
1 byte	`char`	`B`
2 bytes	`int, array`	`D`
3 bytes	`far data pointer`	`X(L), B(H)`
4 bytes	`long`	`D(L), X(H)`

Parameters having a type not listed are passed on the stack (i.e. all those having a size greater than 4 bytes).

Return Value

Function results usually are returned in registers, except if the function returns a result larger than 4 bytes (see below). Depending on the size of the return type, different registers are used:

Size of return value	Type example	Register
1 byte	char	B
2 bytes	int	D
3 bytes	far data pointer	X(L), B(H)
4 bytes	long	D(L), X(H)

Functions returning a result larger than two words are called with an additional parameter. This parameter is the address where the result should get copied to.

Accessing Assembly Variables in an ANSI C Source File

A variable or constant defined in an assembly source file is accessible in an ANSI C source file.

The variable or constant is defined in the assembly source file using the standard assembly syntax.

Variables and constants must be exported using the directive `XDEF` to make them visible from other modules.

Example of Data and Constant Definition:

```
            XDEF    ASMData, ASMConst
DataSec:    SECTION
ASMData:    DS.W    1       ; Definition of a variable
```

```
ConstSec:  SECTION
ASMConst:  DC.W    $44A6  ; Definition of a constant
```

We recommend to generate a header file for each assembler source file. This header file should contain the interface to the assembly module.

An external declaration for the variable or constant must be inserted in the header file.

Example of Data and Constant Declaration:

```
extern int       ASMData;  /* External declaration of a variable */
extern const int ASMConst; /* External declaration of a constant */
```

The variable or constant can then be accessed in the usual way, using their name.

Example of Data and Constant Reference:

```
ASMData = ASMConst + 3;
```

Accessing ANSI C Variables in an Assembly Source File

A variable or constant defined in an ANSI C source file is accessible in an Assembly source file.

The variable or constant is defined in the ANSI C source file using the standard ANSI C syntax.

Example of Data and Constant Definition:

```
unsigned int CData;        /* Definition of a variable */
unsigned const int CConst; /* Definition of a constant */
```

An external declaration for the variable or constant must be inserted in the assembly source file.

This can also be done in a separate file, included in the assembly source file.

Example of Data and Constant Declaration:

```
XREF CData ;  External declaration of a variable

XREF CConst;  External declaration of a constant
```

The variable or constant can then be accessed in the usual way, using their name.

Note: The compiler supports also the automatic generation of assembler include files. See in the compiler manual the description of the compiler option "-la".

Example of Data and Constant Reference:

```
LDAA CConst
....
LDAA CData
....
```

Invoking an Assembly Function in an ANSI C Source File

An function implemented in an assembly source file can be invoked in a C source file. During the implementation of the function in the assembly source file, the programmer should pay attention to the parameter passing scheme of the ANSI C compiler he is using, in order to retrieve the parameter from the right place.

Example of assembler file: mixasm.asm

```
            XREF CData
            XDEF AddVar
            XDEF ASMData

DataSec:    SECTION
ASMData:    DS.B 1
CodeSec:    SECTION
AddVar:
            ADDB CData    ; add CData to the parameter in register B
            STAB ASMData  ; result of the addition in ASMData
            RTS
```

We recommend to generate a header file for each assembler source file. This header file should contain the interface to the assembly module.

```
/* mixasm.h */
#ifndef _MIXASM_H_
#define _MIXASM_H_

void AddVar(unsigned char value);
/* function which adds the paramater value to the global CData */
/* and then stores the result in ASMData */

extern char ASMData;
```

```
/* variable which receives the result of AddVar */
#endif /* _MIXASM_H_ */
```

The function can then be invoked in the usual way, using its name.

Example of C file:

mixc.c (compile it with the compiler option -CC when using the HIWARE Object File Format).

```
static   int Error = 0;
const unsigned char CData=12;
#include "mixasm.h"

void main (void) {
  AddVar(10);
  if (ASMData != CData + 10){
    Error = 1;
  } else {
    Error = 0;
  }
  for(;;);  // wait forever
}
```

Note: Be careful, the assembler will not make any check on the number and type of the function parameters.

The application must be correctly linked.

For these `C` and `.asm` file, a possible linker parameter file is:

Example of linker parameter file: mixasm.prm

```
LINK mixasm.abs
NAMES
  mixc.o mixasm.o
END
SECTIONS
  MY_ROM   = READ_ONLY  0x4000 TO 0x4FFF;
  MY_RAM   = READ_WRITE 0x2400 TO 0x2FFF;
  MY_STACK = READ_WRITE 0x2000 TO 0x23FF;
END
PLACEMENT
  DEFAULT_RAM          INTO MY_RAM;
  DEFAULT_ROM          INTO MY_ROM;
  SSTACK               INTO MY_STACK;
END
INIT main
```

Note: Be careful, use the same memory model and object file format for all the

generated object files.

Support for Structured Types

When option “-Struct: Support for Structured Types” is activated, the macro assembler also supports definition and usage of structured types. This allows an easier way to access ANSI C structured variable in the macro assembler.

In order to provide an efficient support for structured type the macro assembler should provide notation to:

- Define a structured type
- Define a structured variable
- Declare a structured variable
- Access the address of a field inside of a structured variable
- Access the offset of a field inside of a structured variable

Note: Some limitation apply in the usage of the structured type in the macro assembler (See section Limitation below).

Structured Type Definition

The macro assembler will be extended with following new keywords, in order to support ANSI C type definition.

```
STRUCT
UNION
```

The structured type definition can be encoded as:

```
typeName: STRUCT
     lab1: DS.W 1
     lab2: DS.W 1
     ...
   ENDSTRUCT
```

where:

'typeName' is the name associated with the defined type. The type name will be considered as a user define keyword. The macro assembler will be case insensitive on type name.

'STRUCT' specifies that the type is a structured type.

'lab1', 'lab2' are the fields defined inside of the type 'typeName'. The fields

will be considered as user defined labels and the macro assembler will be case sensitive on label names.

As all other directive in assembler, the directives STRUCT and UNION are case insensitive.

The directive STRUCT and UNION cannot start on column 1 and must be preceded by a label.

Type allowed for Structured Type Fields

Field inside of a structured type may be:

- another structured type
- a base type, which can be mapped on 1, 2 or 4 bytes.

The following table shows how the ANSI C standard types are converted in the assembler notation:

ANSI C type	*Assembler Notation*
char	*DS.B*
short	*DS.W*
int	*DS.W*
long	*DS.L*
enum	*DS.W*
bitfield	*-- not supported --*
float	*-- not supported --*
double	*-- not supported --*
data pointer	*DS.W*
function pointer	*-- not supported --*

Variable Definition

The macro assembler should provide a way to define a variable with a specific type. This can be done using following syntax:

```
var: typeName
```

Where

'var' is the name of the variable.

'typeName' is the type associated with the variable.

Example

```
myType: STRUCT
field1:   DS.W 1
field2:   DS.W 1
field3:   DS.B 1
field4:   DS.B 3
field5:   DS.W 1
        ENDSTRUCT

DataSection: SECTION
structVar: TYPE myType ; variable ‘structVar’ is from type ‘myType’
```

Variable Declaration

The macro assembler should provide a way to associated a type with a symbol which is defined externally. This can be done extending the XREF syntax:

```
XREF var: typeName, var2
```

Where

'var' is the name of an externally defined symbol.

'typeName' is the type associated with the variable 'var'.

'var2' is the name of another externally defined symbol. This symbol is not associated with any type.

Example

```
myType: STRUCT
field1:   DS.W 1
field2:   DS.W 1
field3:   DS.B 1
field4:   DS.B 3
field5:   DS.W 1
        ENDSTRUCT

       XREF extData:myType ; variable ‘extData’ is from type ‘myType’
```

Accessing Structured Variable

The macro assembler should provide a way to access each structured type field absolute address and offset.

Accessing a Field Address

To access a structured type field address, the assembler will use the character ':'.

```
var:field
```

Where

'var' is the name of a variable, which was associated with a structured type.

'field' is the name of a field in the structured type associated with the variable.

Example

```
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
field3:    DS.B 1
field4:    DS.B 3
field5:    DS.W 1
         ENDSTRUCT

         XREF myData:myType
         XDEF entry

CodeSec: SECTION
entry:
         LDAA myData:field3 ; Loads register A with the content of
                            ; field field3 from variable myData.
```

Note: The period cannot be used as separator, because in assembler it is a valid character inside of a symbol name.

Accessing a Field Offset

To access a structured type field offset, the assembler will use following notation:

```
<typeName>-><field>
```

Where

'typeName' is the name of a structured type.

'field' is the name of a field in the structured type associated with the variable.

Example

```
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
field3:    DS.B 1
field4:    DS.B 3
field5:    DS.W 1
         ENDSTRUCT
         XREF.B myData
         XDEF   entry

CodeSec: SECTION
entry:
         LDX #myData
         LDAA myType->field3,X ;Adds the offset of field 'field3'
                               ; (4) to X and loads A with the
                               ; content of the pointed address
```

Structured Type: Limitations

Field inside of a structured type may be:

- another structured type
- a base type, which can be mapped on 1, 2 or 4 bytes.

The macro assembler is not able to process bitfields or pointer types.

The type referenced in a variable definition or declaration must be defined previously. A variable cannot be associated with a type defined afterwards.

Make Applications

Assembler Applications

Generating directly an Absolute File

When an absolute file is directly generated by the assembler:

- the application entry point must be specified in the assembly source file using the directive ABSENTRY.
- The whole application must be encoded in a single assembly unit.
- The application should only contain absolute sections.

Generating Object Files

The entry point of the application must be mentioned in the Linker parameter file using the command "`INIT` *funcname*". The application is build of the different object files with the Linker. The Linker is document in a separate document.

Your assembly source files must be separately assembled. Then the list of all the object files building the application must be enumerated in the application PRM file.

Mixed C and assembler Applications

Normally the application starts with the main procedure of a C file. All necessary object files - assembler or C- are linked with the Linker in the same fashion like pure C applications. The Linker is documented in a separate document.

Memory Maps and Segmentation

Relocatable Code Sections are placed in the `DEFAULT_ROM` or `.text` Segment.

Relocatable Data Sections are placed in the `DEFAULT_RAM` or `.data` Segment.

Note: The .text and .data names are only supported when the ELF object file format is used.

There are no checks at all that variables are in a `RAM`. If you mix code and data in a section you can't place the section into `ROM`. That's why we suggest to separate code

and data into different sections.

If you want to place a section in a specific address range, you have to put the section name in the linker parameter file in the placement list.

```
SECTIONS
  ROM1       = READ_ONLY 0x0200 TO 0x0FFF;
  SpecialROM = READ_ONLY 0x8000 TO 0x8FFF;
  RAM        = READ_WRITE 0x4000 TO 0x4FFF;
PLACEMENT
  DEFAULT_ROM   INTO ROM1;
  mySection     INTO SpecialROM;
  DEFAULT_RAM   INTO RAM;
END
```

How To ...

This section covers the following topics:

How To Work with Absolute Sections

An absolute section is a section which start address is known at assembly time.

(See modules `fiboorg.asm` and `fiboorg.prm` in the demo directory)

Defining Absolute Sections in the Assembly Source File

An absolute section is defined using the directive `ORG`. In that case the macro assembler generates a pseudo section, which name is "`ORG_<index>`", where index is an integer which is incremented each time an absolute section is encountered.

Example

Defining an absolute section containing data:

```
        ORG $800   ; Absolute data section.
var:    DS.B    1
        ORG $A00   ; Absolute constant data section.
cst1:   DC.B    $A6
cst2:   DC.B    $BC
```

In the previous portion of code, the label `cst1` will be located at address `$A00`, and label `cst2` will be located at address `$A01`.

```
1    1                      ORG $800
2    2  a000800       var:  DS.B    1
3    3                      ORG $A00
4    4  a000A00 A6    cst1: DC.B    $A6
5    5  a000A01 BC    cst2: DC.B    $BC
```

Defining an absolute section containing code:

```
        XDEF entry
        ORG $C00 ; Absolute code section.
entry:
        LDAA  cst1 ; Load value in cst1
        ADDA  cst2 ; Add value in cst2
        STAA  var  ; Store in var
        BRA  entry
```

In the previous portion of code, the instruction LDAA will be located at address $C00, and instruction ADDA at address $C03.

```
 6   6                          ORG $C00 ; Absolute code section.
 7    7                    entry:
 8    8  a000C00 B6 0A00          LDAA  cst1 ; Load value in cst1
 9    9  a000C03 BB 0A01           ADDA  cst2 ; Add value in cst2
10   10  a000C06 7A 0800            STAA  var  ; Store in var
11   11  a000C09 20F5               BRA  entry
```

In order to avoid problems during linking or execution from an application, an assembly file should at least:

- Initialize the stack pointer if the stack is used.
  The instruction LDS can be used to initialize the stack pointer.
- Publish the application entry point using XDEF.
- The programmer should ensure that the addresses specified in the source file are valid addresses for the MCU being used.

Linking an Application containing Absolute Sections

When the assembler is generating an object file, applications containing only absolute sections must be linked. The linker parameter file must contain at least:

- The name of the absolute file.
- The name of the object file which should be linked.
- The specification of a memory area where the sections containing variables must be allocated. For applications containing only absolute sections, nothing will be allocated there.
- The specification of a memory area where the sections containing code or constants must be allocated. For applications containing only absolute sections, nothing will be allocated there.
- The specification of the application entry point.

- The definition of the reset vector.

The minimal linker parameter file will look as follows:

```
LINK test.abs  /* Name of the executable file generated. */
NAMES
  test.o       /* Name of the object files in the application. */
END
SECTIONS
/* READ_ONLY memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */
  MY_ROM  = READ_ONLY  0x4000 TO 0x4FFF;
/* READ_WRITE memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */
  MY_RAM  = READ_WRITE 0x2000 TO 0x2FFF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  DEFAULT_RAM    INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  DEFAULT_ROM    INTO MY_ROM;
END
INIT entry              /*  Application entry point. */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

Note: There should be no overlap between the absolute section defined in the assembly source file and the memory area defined in the PRM file

Note: As the memory areas (segments) specified in the PRM file are only used to allocate relocatable sections, nothing will be allocated there, when the application contains only absolute sections. In that case you can even specify invalid address ranges in the PRM file

How To Work with Relocatable Sections

A relocatable section is a section which start address is determined at linking time.

(See modules `fibo.asm` and `fibo.prm` in the demo directory)

Defining Relocatable Sections in the Source File

A relocatable section is defined using the directive `SECTION`.

Example

Defining a relocatable section containing data:

```
constSec: SECTION ; Relocatable constant data section.
cst1:     DC.B    $A6
cst2:     DC.B    $BC

dataSec:  SECTION ; Relocatable data section.
var:      DS.B    1
```

In the previous portion of code, the label `cst1` will be located at offset 0 from the section `constSec` start address, and label `cst2` will be located at offset 1 from the section `constSec` start address.

```
     2    2                    constSec: SECTION ; Relocatable
     3    3   000000 A6         cst1:     DC.B    $A6
     4    4   000001 BC         cst2:     DC.B    $BC
     5    5
     6    6                    dataSec:  SECTION ; Relocatable
     7    7   000000           var:      DS.B    1
```

Defining a relocatable section containing code:

```
          XDEF entry
codeSec:  SECTION ; Relocatable code section.
entry:
   LDAA  cst1      ; Load value in cst1
   ADDA  cst2      ; Add value in cst2
   STAA  var       ; Store in var
   BRA  entry
```

In the previous portion of code, the instruction LDAA will be located at offset 0 from the section codeSec start address, and instruction ADDA at offset 3 from the section codeSec start address.

In order to avoid problems during linking or execution from an application, an assembly file should at least:

- Initialize the stack pointer if the stack is used.
  The instruction LDS can be used to initialize the stack pointer.
- Publish the application entry point using the directive XDEF.

Linking an Application containing Relocatable Sections

Applications containing relocatable sections must be linked. The linker parameter file must contain at least:

- The name of the absolute file.
- The name of the object file which should be linked.
- The specification of a memory area where the sections containing variables must be allocated.
- The specification of a memory area where the sections containing code or constants must be allocated.
- The specification of the application entry point
- The definition of the reset vector

The minimal linker parameter file will look as follows:

```
LINK test.abs  /* Name of the executable file generated. */
NAMES
  test.o       /* Name of the object files in the application. */
END
SECTIONS
/* READ_ONLY memory area.  */
  MY_ROM  = READ_ONLY  0x2B00 TO 0x2BFF;
/* READ_WRITE memory area. */
  MY_RAM  = READ_WRITE 0x2800 TO 0x28FF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  DEFAULT_RAM              INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  DEFAULT_ROM, constSec    INTO MY_ROM;
END
INIT entry              /*  Application entry point. */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

Note: The programmer should ensure that the memory ranges he specifies in the SECTIONS block are valid addresses for the controller he is using. Additionally, when using the SDI debugger the addresses specified for code or constant sections must be located in the target board ROM area, otherwise the debugger will not be able to load the application

The module `fibo.asm` located in the demo directory is a small example of usage of relocatable sections in an application.

How To Initialize the Vector Table

The vector table can be initialized in the assembly source file or in the linker parameter file. We recommend to initialize it in the linker parameter file.

Initialize the Vector Table in the PRM File

Initializing the vector table in the Source File using a Relocatable Section

Initializing the vector table in the Source File using an Absolute Section

Initializing the Vector Table in the Linker PRM File

Initializing the vector table from the PRM file allows you to initialize single entries in the table. The user can decide if he wants to initialize all the entries in the vector table or not.

The labels or functions, which should be inserted in the vector table, must be implemented in the assembly source file. All these labels must be published otherwise they cannot be addressed in the linker PRM file.

Example:

```
          XDEF IRQFunc, XIRQFunc, SWIFunc, OpCodeFunc, ResetFunc
DataSec: SECTION
Data:    DS.W 5  ; Each interrupt increments an element in the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
IRQFunc:
          LDAB #0
          BRA  int
XIRQFunc:
          LDAB #2
          BRA  int
SWIFunc:
          LDAB #4
          BRA  int
OpCodeFunc:
          LDAB #6
          BRA  int
ResetFunc:
          LDAB #8
          BRA  entry
int:
          LDX  #Data  ; Load address of symbol Data in X
          ABX  ; X <- address of the appropriate element in the table
          INC  0, X   ; The table element is incremented
          RTI
entry:
          LDS #$AFE
loop:     BRA loop
```

Note: The functions ‘IRQFunc’, ‘XIRQFunc’, ‘SWIFunc’, ‘OpCodeFunc’, ‘Reset-

Func' are published. This is required because they are referenced in the linker PRM file.

Note: As the processor automatically pushes all registers on the stack on occurrence of an interrupt, the interrupt function do not need to save and restore the registers it is using

Note: All Interrupt functions must be terminated with an RTI instruction

The vector table is initialized using the linker command VECTOR ADDRESS.

Example:

```
LINK test.abs
NAMES
  test.o
END

SECTIONS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0B00 TO 0x0CFF;
END
PLACEMENT
  DEFAULT_RAM         INTO MY_RAM;
  DEFAULT_ROM         INTO MY_ROM;
END

INIT ResetFunc
VECTOR ADDRESS 0xFFF2 IRQFunc
VECTOR ADDRESS 0xFFF4 XIRQFunc
VECTOR ADDRESS 0xFFF6 SWIFunc
VECTOR ADDRESS 0xFFF8 OpCodeFunc
VECTOR ADDRESS 0xFFFE ResetFunc
```

Note: The statement 'INIT ResetFunc' defines the application entry point. Usually, this entry point is initialized with the same address as the reset vector.

Note: The statement 'VECTOR ADDRESS 0xFFF2 IRQFunc' specifies that the address of function 'IRQFunc' should be written at address 0xFFF2.

Initializing the Vector Table in the Source File using a Relocatable Section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions, which should be inserted in the vector table must be implemented in the assembler source file or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables.

Example:

```
          XDEF ResetFunc
DataSec: SECTION
Data:    DS.W 5  ; Each interrupt increments an element of the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
IRQFunc:
          LDAB #0
          BRA  int
XIRQFunc:
          LDAB #2
          BRA  int
SWIFunc:
          LDAB #4
          BRA  int
OpCodeFunc:
          LDAB #6
          BRA  int
ResetFunc:
          LDAB #8
          BRA  entry
DummyFunc:
          RTI
int:
          LDX  #Data
          ABX
          INC  0, X
          RTI
entry:
          LDS #$AFE
loop:     BRA loop

VectorTable:SECTION
; Definition of the vector table.
IRQInt:        DC.W IRQFunc
XIRQInt:       DC.W XIRQFunc
SWIInt:        DC.W SWIFunc
OpCodeInt:     DC.W OpCodeFunc
COPResetInt:   DC.W DummyFunc; No function attached to COP Reset.
ClMonResInt:   DC.W DummyFunc; No function attached to Clock
                              ; MonitorReset.
ResetInt     : DC.W ResetFunc
```

Note: Each constant in the section 'VectorTable' is defined as a word (2 Byte constant), because the entries in the vector table are 16 bit wide.

Note: In the previous example, the constant 'IRQInt' is initialized with the address of the label 'IRQFunc'.

Note: In the previous example, the constant 'XIRQInt' is initialized with the address of the label 'XIRQFunc'.

Note: All the labels specified as initialization value must be defined, published (using XDEF) or imported (using XREF) in the assembly source file

The section should now be placed at the expected address. This is performed in the linker parameter file.

Example:

```
LINK test.abs
NAMES  test.o END

SECTIONS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0A00 TO 0x0BFF;
/* Define the memory range for the vector table */
  Vector  = READ_ONLY  0xFFF2 TO 0xFFFF;
END
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
/* Place the section 'VectorTable' at the appropriated address. */
  VectorTable  INTO Vector;
END

INIT ResetFunc
ENTRIES
  *
END
```

Note: The statement 'Vector = READ_ONLY 0xFFF2 TO 0xFFFF' defines the memory range for the vector table.

Note: The statement 'VectorTable INTO Vector' specifies that the section VectorTable should be loaded in the read only memory area Vector. This means, the constant 'IRQInt' will be allocated at address 0xFFF2, the constant 'XIRQInt' will be allocated at address 0xFFF4, and so on. The constant 'ResetInt' will be allocated at address 0xFFFE.

*Note: The statement 'ENTRIES * END' switches smart linking off. If this state-*

ment is missing in the PRM file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file.

Note: When developing a banked application, make sure that the code from the interrupt functions is located in the non banked memory area.

Initializing the Vector Table in the Source File using an Absolute Section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions, which should be inserted in the vector table must be implemented in the assembly source file or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables.

Example:

```
          XDEF ResetFunc
  DataSec: SECTION
  Data:    DS.W 5  ; Each interrupt increments an element of the table.
  CodeSec: SECTION
  ; Implementation of the interrupt functions.
  IRQFunc:
            LDAB #0
            BRA  int
  XIRQFunc:
            LDAB #2
            BRA  int
  SWIFunc:
            LDAB #4
            BRA  int
  OpCodeFunc:
            LDAB #6
            BRA  int
 ResetFunc:
            LDAB #8
            BRA  entry
  DummyFunc:
            RTI
  int:
            LDX  #Data
            ABX
```

```
            INC  0, X
            RTI
    entry:
            LDS #$AFE
    loop:   BRA loop

                  ORG $FFF2
    ; Definition of the vector table in an absolute section starting at
   address
    ;  $FFF2.
    IRQInt:         DC.W IRQFunc
    XIRQInt:        DC.W XIRQFunc
    SWIInt:         DC.W SWIFunc
    OpCodeInt:      DC.W OpCodeFunc
    COPResetInt:    DC.W DummyFunc; No function attached to COP Reset.
    ClMonResInt:    DC.W DummyFunc; No function attached to Clock
                                  ; MonitorReset.
    ResetInt      : DC.W ResetFunc
```

Note: Each constant in the section starting at $FFF2 is defined as a word (2 Byte constant), because the entry in the vector table are 16 bit wide.

Note: In the previous example, the constant 'IRQInt' is initialized with the address of the label 'IRQFunc'.

Note: All the labels specified as initialization value must be defined, published (using XDEF) or imported (using XREF) in the assembly source file.

Note: The statement 'ORG $FFF2' specifies that the following section must start at address $FFF2.

The linker PRM file looks as follows:

Example:

```
LINK test.abs
NAMES
  test.o
END

SECTIONS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0A00 TO 0x0BFF;
END
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
END

INIT ResetFunc
```

```
ENTRIES
  *
END
```

*Note: The statement 'ENTRY * END' switches smart linking off. If this statement is missing in the PRM file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file.*

Note: When developing a banked application, make sure that the code from the interrupt functions is located in the non banked memory area

Splitting an Application into different Modules

Complex application or application involving several programmers can be split into several simple modules. In order to avoid any problem when merging the different modules following rules must be followed:

- For each assembly source file, one include file must be created containing the definition of the symbols exported from this module. For the symbols referring to code label, a small description of the interface is required.

Example of Assembly File (Test1.asm):

```
          XDEF AddSource
          XDEF Source

initStack:EQU $AFF

DataSec: SECTION
Source:   DS.B 1
CodeSec:  SECTION
AddSource:
           ADDA Source
           STAA Source
           RTS
```

Corresponding Include File(Test1.inc):

```
          XREF AddSource
; The function AddSource adds the value stored in the variable
; Source to the content of register A. The result of the computation
; is stored in the variable Source.
;
; Input Parameter: register A contains the value, which should be
;                  added to the variable Source.
```

```
; Output Parameter: Source contains the result of the addition.

        XREF Source
; The variable Source is a byte variable.
```

Example of Assembly File(Test2.asm):

```
        XDEF entry
        INCLUDE "Test1.inc"

initStack: EQU $AFE

CodeSec:   SECTION
entry:     LDS #initStack
           LDAA #$7
           JSR AddSource
           BRA entry
```

The application `.prm` file should list both object files building the application. When a section is present in the different object files, the object file sections are concatenated in a single absolute file section. The different object file sections are concatenated in the order the object files are specified in the `.prm` file.

Example of PRM File(Test2.prm):

```
LINK test2.abs  /* Name of the executable file generated. */

NAMES
  test1.o
  test2.o /*Name of the object files building the application.*/
END

SECTIONS
  MY_ROM  = READ_ONLY  0x2B00 TO 0x2BFF; /* READ_ONLY memory area */
  MY_RAM  = READ_WRITE 0x2800 TO 0x28FF; /* READ_WRITE memory area */
END

PLACEMENT
  DataSec, DEFAULT_RAM INTO MY_RAM;
  /* variables are allocated in MY_RAM */
  CodeSec, ConstSec, DEFAULT_ROM INTO MY_ROM;
  /* code and constants are allocated in MY_ROM */
END
INIT entry       /* Definition of the application entry point. */

VECTOR ADDRESS 0xFFFE entry /* Definition of the reset vector. */
```

Note: The section 'CodeSec' is defined in both object files. In 'test1.o', the section

‘CodeSec’ contains the symbol ‘AddSource’. In ‘test2.o’, the section ‘CodeSec’ contains the symbol ‘entry’. According to the order in which the object files are listed in the NAMES block, the function ‘AddSource’ will be allocated first and symbol ‘entry’ will be allocated next to it.

Using Direct Addressing mode to access Symbols

There are different ways to inform the assembler it should use direct addressing mode on a symbol.

Using Direct Addressing mode to Access External Symbols

External symbols, which should be accessed using the direct addressing mode, must be declared using the directive XREF.B. Symbols which are imported using XREF are accessed using the extended addressing mode.

Example:

```
      XREF.B ExternalDirLabel
      XREF   ExternalExtLabel
…
      LDD   ExternalDirLabel ; Direct addressing mode is used.
…
      LDD   ExternalExtLabel ; Extended addressing mode is used.
```

Using Direct Addressing mode to Access Exported Symbols

Symbols, which are exported using the directive XDEF.B, will be accessed using the direct addressing mode. Symbols which are exported using XDEF are accessed using the extended addressing mode.

Example:

```
      XDEF.B DirLabel
      XDEF   ExtLabel
…
      LDD   DirLabel ; Direct addressing mode is used.
…
      LDD   ExtLabel ; Extended addressing mode is used.
```

Defining Symbols in the Direct Page

Symbols, which are defined in the predefined section BSCT are always accessed using direct addressing mode.

Example:

```
…
     BSCT
DirLabel: DS.B 3
dataSec: SECTION
ExtLabel: DS.B 5
…
codeSec: SECTION
…
     LDD   DirLabel ; Direct addressing mode is used.
…
     LDD   ExtLabel ; Extended addressing mode is used.
```

Using Force Operator

A force operator can be specified in an assembly instruction to force direct or extended addressing mode.

The supported force operators are:

- < or .B to force direct addressing mode
- > or .W to force extended addressing mode.

Example:

```
…
dataSec: SECTION
label: DS.B 5
…
codeSec: SECTION
…
     LDD   <label ; Direct addressing mode is used.
     LDD   label.B; Direct addressing mode is used.
…
     LDD   >label ;  Extended addressing mode is used.
     LDD   label.W ; Extended addressing mode is used.
```

Using SHORT Sections

Symbols, which are defined in a section defined with the qualifier SHORT are

always accessed using the direct addressing mode.

Example:

```
…
shortSec:SECTION SHORT
DirLabel: DS.B 3
dataSec: SECTION
ExtLabel: DS.B 5
…
codeSec: SECTION
…
     LDD   DirLabel ; Direct addressing mode is used.
…
     LDD   ExtLabel ; Extended addressing mode is used.
```

Assembler Messages

There are five kinds of messages generated by the assembler:

DISABLED:
Disabled messages are not printed unless they are explicitly enabled.

INFORMATION:
A message will be printed and assembling will continue. Information messages are used to inform the user about various topics.

WARNING:
A message will be printed and assembling will continue. Warning messages are used to indicate possible programming errors to the user.

ERROR:
A message will be printed and assembling will be stopped. Error messages are used to indicate illegal usage of the language.

FATAL:
A message will be printed and assembling will be aborted. A fatal message indicates a severe error which anyway will stop the assembling.

If the assembler prints out a message, the message contains a message code ('A' for Assembler) and a decimal number. This number may be used to search very fast for the indicated message.

All messages generated by the assembler are documented in increasing number order for easy and fast retrieval.
Each message also has a description and, if available, a short example with a possible solution or tips to fix a problem.

For each message the type of the message is also noted, e.g. [ERROR] indicates that the message is an error message.

A1: Unknown message occurred

[FATAL]

Description
The application tried to emit a message which was not defined. This is a internal error which should not occur. Please report any occurrences to you distributor.

Tips
Try to find out the and avoid the reason for the unknown message.

A2: Message overflow, skipping <kind> messages

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The application did show the number of messages of the specific kind as controlled with the options -WmsgNi, -WmsgNw and -WmsgNe. Further options of this kind are not displayed.

Tips
Use the options -WmsgNi, -WmsgNw and -WmsgNe to change the number of messag-

es.

A50: Input file '<file>' not found

[FATAL]

Description

The Application was not able to find a file needed for processing.

Tips

Check if the file really exits. Check if you are using a file name containing spaces (in this case you have to quote it).

A51: Cannot open statistic log file <file>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

It was not possible to open a statistic output file, therefore no statistics are generated. Note: Not all tools do support statistic log files. Even if a tool does not support it, the message does still exist, but is never issued in this case, of course.

A52: Error in command line <cmd>

[FATAL]

Description

In case there is an error while processing the command line, this message is issued.

A64: Line Continuation occurred in <FileName>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In any environment file, the character '\' at the end of a line is taken as line continuation. This line and the next one are handles as one line only. Because the path separation character of MS-DOS is also '\', paths are often incorrectly written ending with '\'. Instead use a '.' after the last '\' to not finish a line with '\' unless you really want a line continuation.

Example

Current Default.env:

```
...
LIBPATH=c:\metrowerks\lib\
OBJPATH=c:\metrowerks\work
...
```

Is taken identical as

```
...
LIBPATH=c:\metrowerks\libOBJPATH=c:\metrowerks\work
...
```

Tips

To fix it, append a '.' behind the '\'

```
...
LIBPATH=c:\metrowerks\lib\.
OBJPATH=c:\metrowerks\work
...
```

Note:

Because this information occurs during the initialization phase of the application, the message prefix might not occur in the error message. So it might occur as "64: Line Continuation occurred in <FileName>".

A65: Environment macro expansion error '<description>' for <variablename>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

During a environment variable macro substitution an problem did occur. Possible causes are that the named macro did not exist or some length limitation was reached. Also recursive macros may cause this message.

Example

Current Default.env:

```
...
LIBPATH=${LIBPATH}
...
```

Tips

Check the definition of the environment variable.

A66: Search path <Name> does not exist

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The tool did look for a file which was not found. During the failed search for the file, a non existing path was encountered.

Tips

Check the spelling of your paths.
Update the paths when moving a project.
Use relative paths in your environment variables.
Check if network drives are available

A1000: Conditional directive not closed

[ERROR]

Description

One of the conditional blocks is not closed. A conditional block can be opened using one of the following directives:
IF, IFEQ, IFNE, IFLT, IFLE, IFGT, IFGE, IFC, IFNC, IFDEF, IFNDEF.

Example

```
    IFEQ (defineConst)
const1:  DC.B 1
const2:  DC.B 2
```

Tips

Close the conditional block with an ENDIF or ENDC directive.

Example

```
    IFEQ (defineConst)
const1:  DC.B 1
const2:  DC.B 2
    ENDIF
```

Be careful:

A conditional block, which starts inside of a macro, must be closed within the same macro.

Example

The following portion of code generates an error, because the conditional block “IFEQ” is opened within the macro “MyMacro” and is closed outside from the macro.

```
MyMacro: MACRO
    IFEQ (SaveRegs)
         DC.B 1
```

```
        DC.B 1
        ENDM
        DC.B 1
    ENDIF
```

A1001: Conditional else not allowed here

[ERROR]

Description

A second ELSE directive is detected in a conditional block.

Example

```
    IFEQ (defineConst)
  ...
    ELSE
  ...
    ELSE
  ...
    ENDIF
```

Tips

Remove the superfluous ELSE directive.

Example

```
    IFEQ (defineConst)
  ...
    ELSE
  ...
    ENDIF
```

A1002: CASE, DEFAULT or ENDSW detected outside from a SWITCH block

[ERROR]

Description

In Avocet compatibility mode, a CASE, DEFAULT or ENDSW directive was found without a previous SWITCH directive.

Note: This message does only occur for assemblers supporting the avocet compatibility mode.

Example

```
   xxx:   equ 0

   ;SWITCH xxx
     CASE 1
        DC.B 100
     CASE 2
        DC.B 200
     CASE 4
        DC.B 400

     DEFAULT
        FAIL 1
   ENDSW
```

Tips

Remove the semicolon in the example.

Make sure that your assembler does support the avocet compatibility mode and that this mode is switched on.

A1003: CASE or DEFAULT is missing

[ERROR]

Description

In Avocet compatibility mode, after a SWITCH directive, an expression other than a CASE or DEFAULT entry was found.

Note: This message does only occur for assemblers supporting the avocet compatibility mode.

Example

```
xxx:   equ 0

SWITCH xxx
  ; CASE 1
    DC.B 0
  CASE 2
     DC.W 0
  CASE 4
     DC.L 0

  DEFAULT
     FAIL 1
ENDSW
```

Tips

Remove the semicolon in the example.

Make sure that your assembler does support the avocet compatibility mode and that this mode is switched on.

A1004: Macro nesting too deep. Possible recursion? Stop processing. (Set level with -MacroNest)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The macro expansion level was below the limit configured with the option -MacroNest.

Example

In the following example, “\2” was used instead of the indented “/2”. “\2” is taken by the assembler as second argument, which is not present and therefore it is replaced with the empty argument. Therefore this example leads to an endless macro recursion.

```
X_NOPS:    MACRO
  \@NofNops: EQU \1
          IF \@NofNops >= 1
             IF \@NofNops == 1
                NOP
             ELSE
                X_NOPS \@NofNops\2
                X_NOPS \@NofNops-(\@NofNops\2)
             ENDIF
          ENDIF
        ENDM

        X_NOPS 17
```

Tips

Use the option -MacroNest to configure the macro expansion level.

In the above example, use /2 to get the correct macro:

```
X_NOPS:    MACRO
```

```
\@NofNops: EQU \1
        IF \@NofNops >= 1
           IF \@NofNops == 1
              NOP
           ELSE
              X_NOPS \@NofNops/2
              X_NOPS \@NofNops-(\@NofNops/2)
           ENDIF
        ENDIF
      ENDM

      X_NOPS 17
```

See also
Option -MacroNest

A1051: Zero Division in expression

[DISABLE, INFORMATION, WARNING, ERROR]

Description
A zero division is detected in an expression.

Example
```
label:  EQU 0
label2: EQU $5000
        DC (label2/label)
```

Tips
Modify the expression or specify it in a conditional assembly block.

Example
```
label:  EQU 0
label2: EQU $5000
      IFNE (label)
        DC  (label2/label)
      ELSE
        DC  label2
      ENDIF
```

A1052: Right parenthesis expected

[ERROR]

Description
A right parenthesis is missing in an assembly expression.

Example
```
variable: DS.W 1
label1: EQU (2*4+6
label3: EQU LOW(variable
label4: EQU HIGH(variable
```

Tips
Insert the right parenthesis at the correct position.

Example
```
variable: DS.W 1
label1: EQU (2*4+6)
label3: EQU LOW(variable)
label4: EQU HIGH(variable)
```

A1053: Left parenthesis expected

[ERROR]

Description

A left parenthesis is missing in an assembly expression.

Example

```
variable: ds.w 1

label1: EQU LOW variable)
label2: EQU HIGH variable)
```

Tips

Insert the left parenthesis at the correct position.

Example

```
label1: EQU LOW(variable)
label2: EQU HIGH(variable)
```

A1054: References on non-absolute objects are not allowed when options -FA1 or -FA2 are enabled

[ERROR]

Description

A reference to a relocatable object has been detected during generation of an absolute file by the assembler.

Example

```
         XREF extData
DataSec: SECTION
data1:   DS.W 1
         ORG $800
entry:
         DC.W extData
         DC.W data1+2
```

Tips

When you are generating an absolute file, your application should be encoded in a single source file, and should only contain any relocatable symbol.
So in order to avoid this message, define all your section as absolute section and remove all XREF directives from your source file.

Example

```
         ORG $B00
data1:   DS.W 1
         ORG $800
entry:
         DC.W data1+2
```

A1055: Error in expression

[ERROR]

Description

An error has been discovered in an expression while parsing it.

```
Example
CodeSec2: SECTION
Entry2:
          LDAA   #$08
label:    JMP   (Entry2 + 1
```

Tips
Correct the expression.

A1056: Error at end of expression

[DISABLE, INFORMATION, WARNING, ERROR]

Description
An error has been detected by the assembler at the end of the read expression.

Example
```
char: SET 1 this is a comment
```

Tips
Remove the not correct symbol at the end of line or insert a comment start ";".

Example
```
char: SET 1 ;this is a comment
```

A1057: Cutting constant because of overflow

[DISABLE, INFORMATION, WARNING, ERROR]

Description
A constant was cutted because of an overflow. Only the lower bits were used to generate the output.

Example
```
DC $123456789
```

Tips
Only use 32 bit constants. Use several DC's to produce larger values.

A1058: Illegal floating point operation

[DISABLE, INFORMATION, WARNING, ERROR]

Description
An illegal floating point operation other than unary minus or unary plus has been detected.

A1059: != is taken as EQUAL

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The != operator is taken as equal. This behavior is different from the C language or the usual assembler behavior. The behavior is caused by the Option -Compat. Disable the message, if you are aware of the different semantic

See also
Option -Compat

A1060: Implicit comment start

[DISABLE, INFORMATION, WARNING, ERROR]

Description
With the alternate comment syntax of the option Option -Compat=C, this message is issued if the ignored part does not start with a star ("*") or with a semicolon (";").

See also
Option -Compat

A1061: Floating Point format is not supported for this case

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The floating point value is not supported at this place.

A1062: Floating Point number expected

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The assembler did expect a floating point value, but he found an expression of a different type.
Note: Not all assemblers do support floating point constants. Assemblers not supporting floating point do not issue this message.

Example

```
; The example only works with assemblers supporting floating point with a dc.f directive
label
 dc.f label
```

A1101: Illegal label: label is reserved

[ERROR]

Description

A reserved identifier is used as label. Reserved identifiers are the mnemonics associated with target processor registers and some additional Reserved Symbols.

Example

```
X: SET 3
```

Tips

Modify the name of the label to a identifier which is not reserved.

Example

```
_X: SET 3
```

See also

Reserved Symbols

A1103: Illegal redefinition of label

[ERROR]

Description

The label specified in front of a comment or an assembly instruction or directive, is detected twice in a source file.

Example

```
DataSec1: SECTION
label1: DS.W 2
label2: DS.L 2
…
CodeSec1: SECTION
Entry:  LDS #$4000
        LDX #label1
        CPX #$500
        BNE label2
…
label2: RTS
```

Tips

Modify the label names, in order to have unique label identification in each assembly file.

Example:

```
DataSec1: SECTION
DataLab1: DS.W 2
DataLab2: DS.L 2
```

```
…
CodeSec1: SECTION
Entry:  LDS #$4000
        LDX #DataLab1
        CPX #$500
        BNE CodeLab2
…
CodeLab2: RTS
```

A1104: Undeclared user defined symbol: <symbolName>

[ERROR]

Description

The label <symbolName> is referenced in the assembly file, but it is never defined.

Example

```
Entry:
  LDX #56
  STX #Variable
  RTS
```

Tips

The label <symbolName> must be either defined in the current assembly file or specified as an external label.

Example:

```
  XREF  Variable
  …
Entry:
  LDX #56
  STX #Variable
  RTS
```

A1201: Label <labelName> referenced in directive ABSENTRY. Only labels defined in a code segment are allowed in the ABSENTRY directive

[ERROR]

Description

The label specified in the directive ABSENTRY is an EQU label or is located in a data section. The label specified in ABSENTRY must be a valid label defined in a code section.

Example

```
        ABSENTRY  const
const:  EQU     $1000
        ORG     const
        DC.B    1
        DC.B    2
```

Tips

Specify a label defined in a code section in ABSENTRY or remove the directive ABSENTRY.

Example

```
        ABSENTRY  entry
const:  EQU     $1000
        ORG     const
```

```
entry:  DC.B    1
        DC.B    2
```

A1251: Cannot open object file: Object file name too long

[ERROR]

Description

The object file is derived from the source file name by changing the extension to ".o". If the source file name is extremely long, then this may fail.

Tips

Use shorter filenames.

A1252: The exported label <name> is using an ELF extension

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is only issued when using the ELF object file format. I can be ignored when using the SmartLinker, however, foreign linker may not know this extension and therefore the linking might fail.
The exported label <name> is using an ELF extension for exported labels, which are defined as imported label plus offset. This situation cannot be expressed in a standard ELF symbol table, so the assembler is generating a symbol with type STT_LOPROC. This message is disabled by default, so it does not occur unless it is explicitly enabled. When setting this message to an error, code containing such cases cannot be assembled.

Example

```
XREF ImportedLabel
ExportedLabel: EQU ImportedLabel + 1
XDEF ExportedLabel
```

Tips

Set this message to an error when you plan to use a foreign linker. Adapt the source code so that this case does not occur.

A1253: Limitation: code size > <SizeLimit> bytes

[ERROR]

Description

The assembler is running in demo mode and the code size limitation was reached. Therefore the assembly process is stopped.

Tips

Make sure the license is correctly installed.
Check the about box about the current license state.

A1301: Structured type redefinition: <TypeName>

[ERROR]

Description

The same name has been associated with two different structured types.
<TypeName> is the name of the structured type, which is defined twice.
Note: Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

Example

```
myType:   STRUCT
field1:     DS.W 1
field2:     DS.W 1
```

```
          ENDSTRUCT

          XREF myData:myType

myType:   STRUCT
field3:     DS.B 1
field4:     DS.B 3
          ENDSTRUCT
```

Tips

Change the name of one of the structured type.

Example

```
myType1:  STRUCT
field1:     DS.W 1
field2:     DS.W 1
          ENDSTRUCT

          XREF myData:myType1

myType2:  STRUCT
field3:     DS.B 1
field4:     DS.B 3
          ENDSTRUCT
```

A1302: Type <TypeName> is previously defined as label

[ERROR]

Description

The identifier used to identify a structured type was previously used as a label. <TypeName> is the name of the structured type, which is already used as label name. Note: Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

Example

```
myType:  DS.W 3
...
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
         ENDSTRUCT
```

Tips

Change the name of one of the structured type or of the label .

Example

```
myVar:   DS.W 3
...
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
         ENDSTRUCT
```

A1303: No type defined

[ERROR]

Description

A directive only allowed inside of s structured type definition was found without a leading STRUCT or UNION.

Note: Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

Example

```
field1:    DS.W 1
field2:    DS.W 1
         ENDSTRUCT
```

Tips

Check the STRUCT directive at the start.

Example

```
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
         ENDSTRUCT
```

A1304: Field <FieldName> is not declared in specified type

[ERROR]

Description

The field specified is not part of the structured type associated with the variable addressed.
<FieldName> is the name of the field addressed in the variable.
Note: Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

Example

```
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
         ENDSTRUCT

         XREF myData:myType
         XDEF entry
CodeSec: SECTION
entry:
         NOP
         NOP
         LDX myData:field33
```

Tips

Change the name of the field to an existing field or define the field in the structured type.

Example:

```
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
         ENDSTRUCT

         XREF myData:myType
         XDEF entry
CodeSec: SECTION
entry:
         NOP
         NOP
         LDX myData:field2
```

A1305: Type name expected

[ERROR]

Description

The symbol specified after a TYPE directive is not a previous defined structured type. Note: Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

Example

```
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
         ENDSTRUCT

DataSec: SECTION
myData:  TYPE yType
         XDEF entry
CodeSec: SECTION
entry:
         NOP
         NOP
         LDX myData:field2
```

Tips

Change the name of the type for a valid type name.

Example:

```
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
         ENDSTRUCT

DataSec: SECTION
myData:  TYPE myType
         XDEF entry
CodeSec: SECTION
entry:
         NOP
         NOP
         LDX myData:field2
```

A1401: Value out of range -128..127

[ERROR]

Description

The offset between the current PC and the label specified as PC relative address is not in the range of a signed byte (smaller than -128 or bigger than 127). An 8 bit signed PC relative offset is expected in following instructions:

- Branch instructions
  BCC, BCS, BEQ, BGE, BGT, BHI, BHS, BLE, BLO, BLS, BLT, BMI, BNE, BPL, BRA, BRN, BSR, BVC, BVS
- Third operand in following instructions:
  BRCLR, BRSET

Example for branch instruction

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION
         …
         LDD  var1
         BNE  label
dummyBl: DCB.B 200, $A7
label    STD var2
```

Tips

If you have used one of the branch instructions, use the corresponding long-branch instruction.

Example:

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION
         …
         LDD  var1
         LBNE label
dummyBl: DCB.B 200, $A7
label    STD var2
```

Example for BRCLR instruction

```
DataSec: SECTION
var1:    DS.W 100
CodeSec: SECTION
         …
         LDX  #var1
         BRCLR  3, X, #$05, label
dummyBl: DCB.B 200, $A7
label    STD var2
```

Tips

If you have used a BRSET or BRCLR, you should replace the BRCLR instruction by following sequence of code:

```
         LDAB  <first operand in the BRCLR>
         ANDB  <second operand in BRCLR>
         LBEQ  <third operand in BRCLR>
```

Example:

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION
         …
         LDX  #var1
         LDAB 3, X
         ANDB #$05
         LBEQ label
dummyBl: DCB.B 200, $A7
label    STD var2
```

A1402: Value out of range -32768..32767

[ERROR]

Description

The offset between the current PC and the label specified as PC relative address is not in the range of a signed word (smaller than -32768 or bigger than 32767).
Note: Not all assemblers do have instructions with 16 bit PC relative addressing mode. Such assemblers will not issue this message at all.
A 16 bit signed PC relative offset is expected in following instructions:

- Long-branch instructions
  LBCC, LBCS, LBEQ, LBGE, LBGT, LBHI, LBHS, LBLE, LBLO, LBLS, LBLT, LBMI, LBNE, LBPL, LBRA, LBRN, LBSR, LBVC, LBVS

Example

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION
         …
         LDD  var1
         LBNE label
dummyBl: DCB.B 20000, $A7
         DCB.B 20000, $A7
label    STD var2
```

Tips

Replace the long-branch instruction by following sequence of code:

```
        <Inverse branch instruction> label1
        JMP  label
label1:
```

Example:

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION
         …
         LDD  var1
         BEQ  label1
         JMP  label
label1:
dummyBl: DCB.B 40000, $A7
label    STD var2
```

A1405: PAGE with initialized RAM not supported

[ERROR]

Description

The Macro Assembler does not support the use of the HIGH operator with initialized RAM in the HIWARE format.
In the ELF format, it is allowed and this message is not used.
Note: not all assemblers do support the PAGE operator.

Example

```
cstSec:  SECTION
pgEntry   DC.B PAGE(entry)
adrEntry: DC.W entry
codeSec: SECTION
entry:
```

```
        NOP
        NOP
```

Tips

You can load the whole address from the entry label using a DC.L directive. The only draw back is that you have allocated 4 byte to store the address instead of 3 bytes.

Example

```
cstSec:  SECTION
adrEntry: DC.L entry
codeSec: SECTION
entry:
        NOP
        NOP
```

A1406: HIGH with initialized RAM not supported

[ERROR]

Description

The Macro Assembler does not support the use of the HIGH operator with initialized RAM in the HIWARE format.

In the ELF format, it is allowed and this message is not used.
Note: not all assemblers do support the HIGH operator.

Example

```
MyData: SECTION
table:  DS.W 1
        DC.B high(table)
```

A1407: LOW with initialized RAM not supported

[ERROR]

Description

The Macro Assembler does not support the use of the LOW operator with initialized RAM in the HIWARE format.

In the ELF format, it is allowed and this message is not used.
Note: not all assemblers do support the LOW operator.

Example

```
MyData: SECTION
table:  DS.W 1
        DC.B low(table)
```

A1408: Out of memory, Code size too large

[ERROR]

Description

The assembler runs out of memory because of a very large section.
Note: This assembler version does no longer have the 32k size limitation of previous versions.

A1410: EQU or SET labels are not allowed in a PC Relative addressing mode.

[ERROR]

Description

An absolute EQU or SET label has been detected in an indexed PC relative addressing mode.
This is not legal in a relocatable expression.
Note: Not all assemblers do have special PC Relative addressing modes. Such assem-

blers will not issue this message at all.

Example

```
label:      EQU $FF30
dataSec:    SECTION
data:       DS.W 1
codeSec1:   SECTION
entry:
            LDD  label, PCR
            STD  data
```

Tips

Make the section an absolute section.
Example of Merging sections:

```
label:      EQU $FF30
dataSec:    SECTION
data:       DS.W 1
            ORG $C000
entry:
            LDD  label, PCR
            STD  data
```

A1411: PC Relative addressing mode is not supported to constants

[ERROR]

Description

An absolute expression has been detected in an indexed PC relative addressing mode. This is not legal in a relocatable expression.
Not all assemblers do have special PC Relative addressing modes. Such assemblers will not issue this message at all.

Example

```
dataSec:    SECTION
data:       DS.W 1
codeSec1:   SECTION
entry:
            LDD  $FF35, PCR
            STD  data
```

Tips

Make the section an absolute section.
Example of Merging sections:

```
dataSec:    SECTION
data:       DS.W 1
            ORG $C000
entry:
            LDD  $FF35, PCR
            STD  data
```

A1412: Relocatable object <Symbol> not allowed if generating absolute file

[Error]

Description

No relocatable objects are allowed if the user requests the generation of an absolute file. This message occurs primarily for objects in the default (relocatable) section.

Example

```
      ABSENTRY main
```

```
main: DC.B 1
      DC.B 2
```

Tips
Place all objects into absolute sections.

Example

```
      ABSENTRY main
      ORG  $1000
main: DC.B 1
      DC.B 2
```

A1413: Value out of relative range

[Disabled, Information, Warning, Error]

Description
Some value did not fit into the operand field of an instruction. This message can be disabled if the value should be just truncated.

Tips
Check if you can place the code and the referenced object closer together. Try to generate a smaller displacement. If this is not possible, consider using another instruction or addressing mode.

A1414: Cannot set fixup to constant

[Error]

Description
The assemble cannot set a fixup because the referenced object is just a constant rather then an object. One case when the assembler must generate a fixup are PCR relative accesses in relocatable code. Then the assembler does need an object which refers to the accessed address.

Tips
Check why the assembler has to set a fixup instead of just using a constant.

A1415: Cutting fixup overflow

[Disabled, Information, Warning, Error]

Description
A constant value does not fit into a field and is therefore cutted.

Example

```
       DC.B Label+1
Label: EQU $ff
       DC.B Label+1
```

Tips
Use a larger field, if necessary.

```
       DC.W Label+1
Label: EQU $ff
       DC.W Label+1
```

A1416: Absolute section starting at <Address> size <Size> overlaps with absolute section starting at <Address>

[Disabled, Information, Warning, Error]

Description
Two absolute sections are overlapping each other.

Example

```
  ORG $1000
  DC.B 0,1,2,3
```

```
  ; address $1004

DA: SECTION
  DC.B 1

  ORG $1001
  DC.B 0,1,2,3
  ; address $1005
```

Tips

Use non overlapping areas, whenever possible.
Use relocatable sections if you want to split up a memory area into several modules.
Calculate the start address of the second with the end address of the first, if they are in the same assembly unit.
Example

```
  ORG $1000
  DC.B 0,1,2,3
  ; address $1004
SectEnd: EQU *

DA: SECTION
  DC.B 1

  ORG SectEnd
  DC.B 0,1,2,3
  ; address $1008
```

A1417: Value out of possible range

[Disabled, Information, Warning, Error]

Description

A constant value does not fit into a field. This message is used to stop the assembly for some fixup overflow cases.
Tips:
Usually this message is used for branch distances, if so, try to use a branch with a larger range.

A1502: Reserved identifiers are not allowed as instruction or directive

[ERROR]

Description

The identifier detected in an assembly line instruction part is a Reserved Symbol.

See also

Reserved Symbols

A1503: Error in option -D: <Description>

[Disabled, Information, Warning, Error]

Description

An option -D was used with illegal content. The format for -D is “-D” name [“=”value]. The name must be a legal for a label. The value must be a number. There must be a number after an equal (“=”).

Example

Not a legal label name:

```
-D1
```

After a =, the there must be a value:

```
-DLabelName=
```

Unexpected text at the end:
`-D"LabelName1=1 1"`

See also
Option -D

A1601: Label must be terminated with a ":"

[ERROR]

Description
This message is issued only when labels must be terminated with a colon. For some targets, this is not required. Then this message is not issued.
This message is only generated when the MCUasm compatibility is switched on. In this case, all labels must be terminated with a semi-colon (:) character.

A1602: Invalid character at end of label (<LabelName>): semicolon or space expected

[ERROR]

Description
The specified label is terminated by an invalid character. The following characters are allowed in a label:

- All alphabetical characters ('a'.. 'z', 'A', 'Z').
- All numerical characters ('0' .. '9').
- '.' and '_'.

<LabelName> is the name of the wrong label detected (including the invalid character).

Example
```
Data1#    DS.B 1
Data2#6   DS.B 1
```

Tips
Remove the invalid character or replace it by a '_' .

Example
```
Data1     DS.B 1
Data2_6   DS.B 1
```

A1603: Directive, instruction or macro name expected: <SymbolName> detected

[ERROR]

Description
The symbol detected in the operation field is not a valid directive, instruction or macro name.
<SymbolName> is the name of the invalid string detected in the operation field.

Example
```
label:    XXX 3
label2:   label
```

Tips
Replace the specified symbolName by a valid instruction, directive or macro name.

A1604: Invalid character detected at the beginning of the line: <Character>

[ERROR]

Description
The character detected on column 1 is not valid. For the macro assembler everything starting on column 1 is supposed to be a label. The following characters are allowed at

the beginning of a label:
- All alphabetical characters (‘a’.. ‘z’, ‘A’, ‘Z’).
- ‘.’ and ‘_’.

<Character> is the character detected on column 1.

Example
```
@label:     DS.B 1
4label2:    DS.B 2
```
Tips
Replace the specified character by a ‘.’ or a ‘_’.

Example
```
_label:     DS.B 1
.label2:    DS.B 2
```
A1605: Invalid label name: <LabelName>
[ERROR]

Description
The character detected at the beginning of a label is not valid. The following characters are allowed at the beginning of a label:
- All alphabetical characters (‘a’.. ‘z’, ‘A’, ‘Z’).
- ‘.’ and ‘_’.

<LabelName> is the label name detected.

Example
```
#label:     DS.B 1
```
Tips
Replace the specified character by a ‘.’ or a ‘_’.

Example
```
_label:     DS.B 1
```
A2301: Label is missing
[ERROR]

Description
A label name is missing on the front of an assembly directive requiring a label. These directives are:
SECTION, EQU, SET

Example
```
  SECTION 4
  …
  EQU $67
  …
  SET $77
```
Tips
Insert a label in front of the directive.

Example
```
codeSec: SECTION 4
  …
myConst: EQU $67
  …
mySetV:  SET $77
```
A2302: Macro name is missing
[ERROR]

Description
A label name is missing on the front of a MACRO directive.

Example
```
MyData:     SECTION
Data1:      DS.B 1
            MACRO
            DC.B \1
            ENDM

MyCode:     SECTION
Entry:
```

Tips
Insert a label in front of the MACRO directive.

Example
```
MyData:     SECTION
Data1:      DS.B 1
allocChar:  MACRO
            DC.B \1
            ENDM

MyCode:     SECTION
Entry:
```

A2303: ENDM is illegal

[ERROR]

Description
A ENDM directive is detected outside of a macro.

Example
```
MyData:     SECTION
Data1:      DS.B 1
allocChar:  MACRO
            DC.B \1
            ENDM

MyCode:     SECTION
Entry:
            ENDM
```

Tips
Remove the superfluous ENDM directive.

Example
```
MyData:     SECTION
Data1:      DS.B 1
allocChar:  MACRO
            DC.B \1
            ENDM

MyCode:     SECTION
Entry:
```

A2304: Macro definition within definition

[ERROR]

Description
A macro definition is detected inside of another macro definition. The macro assembler does not support this.

Example
```
allocChar: MACRO
allocWord: MACRO
             DC.W \1
           ENDM
             DC.B \1
           ENDM
```

Tips
Define the second macro outside from the first one.

Example
```
allocChar: MACRO
             DC.B \1
           ENDM
allocWord: MACRO
             DC.W \1
           ENDM
```

A2305: Illegal redefinition of instruction or directive name

[ERROR]

Description
An assembly directive or a mnemonic has been used as macro name. This is not allowed to avoid any ambiguity when the symbol name is encountered afterward. The macro assembler cannot detect if the symbol refers to the macro or the instruction.

Example
```
DC: MACRO
      DC.B \1
    ENDM
```

Tips
Change the name of the macro to an unused identifier.

Example
```
allocChar: MACRO
             DC.B \1
           ENDM
```

A2306: Macro not closed at end of source

[ERROR]

Description
An ENDM directive is missing at the end of a macro. The end of the input file is detected before the end of the macro.

Example
```
allocChar: MACRO
             DC.B \1
myData:    SECTION SHORT
char1:     DS.B 1
char2:     DS.B 1
myConst:   SECTION SHORT
init1:     DC.B $33
init2:     DC.B $43
```

```
....
```

Tips

Insert the missing ENDM directive at the end of the macro.

Example

```
allocChar: MACRO
             DC.B \1
           ENDM
myData:    SECTION SHORT
char1:     DS.B 1
char2:     DS.B 1
myConst:   SECTION SHORT
init1:     DC.B $33
init2:     DC.B $43
....
```

A2307: Macro redefinition

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The input file contains the definition of two macros, which have the same name.

Example

```
alloc: MACRO
         DC.B \1
       ENDM
alloc: MACRO
         DC.W \1
       ENDM
```

Tips

Change the name of one of the macros to generate unique identifiers.

Example

```
allocChar: MACRO
             DC.B \1
           ENDM
allocWord: MACRO
             DC.W \1
           ENDM
```

A2308: File name expected

[ERROR]

Description

A file name is expected in an INCLUDE directive.

Example

```
     INCLUDE 1234
```

Tips

Specify a file name after the include directive.

Example

```
     INCLUDE "1234" ; file is named "1234"
```

A2309: File not found

[ERROR]

Description

The assembler cannot find the file, which name is specified in the include directive.

Tips

If the file exist, check if the directory where it is located is specified in the GENPATH environment variable.
First check if your project directory is correct. A file “default.env” should be located there, where the environment variables are stored.
The macro assembler looks for the included files in the working directory and then in the directory enumerated in the GENPATH environment variable.
If the file do not exist, create it or remove the include directive.

A2310: Size specification expected

[ERROR]

Description

An invalid size specification character is detected in a DCB, DC, DS, FCC, FCB, FDB, FQB, RMB, XDEF or XREF, directive.
For XDEF and XREF directives, valid size specification characters are:

- .B: for symbols located in a section where direct addressing mode can be used.
- .W: for symbols located in a section where extended addressing mode must be used.

For DCB, DC, DS, FCC, FCB, FDB, FQB and RMB directives, valid size specification characters are:

- .B: for Byte variables.
- .W: for Word variables.
- .L: for Long variables.

Example

```
DataSec: SECTION
label1: DS.Q 2

ConstSec: SECTION
label2: DC.I 3, 4, 66
```

Tips

Change the size specification character to a valid one.

Example

```
DataSec: SECTION
label1: DS.W 2

ConstSec: SECTION
label2: DC.W 3, 4, 66
```

A2311: Symbol name expected

[ERROR]

Description

A symbol name is missing after a XDEF, XREF, IFDEF or IFNDEF directive.

Example

```
         XDEF $5645
         XREF ; This is a comment
CodeSec: SECTION

         IFDEF $5634
         ENDIF
```

Tips

Insert a symbol name at the requested position.

Example

```
            XDEF exportedSymbol
            XREF importedSymbol; This is a comment
CodeSec:    SECTION

exportedSymbol:
            IFDEF changeBank
            ENDIF
```

A2312: String expected

[ERROR]

Description

A character string is expected at the end of a FCC, IFC or IFNC directive.

Example

```
one:        MACRO
            IFC \1,""
              DS.B 1
            ELSE
              DC.B \1
            ENDIF
            ENDM
            one  $42
```

Tips

Insert a character string at the requested position.

Example

```
one:        MACRO
            IFC "\1",""
              DS.B 1
            ELSE
              DC.B \1
            ENDIF
            ENDM
            one  $42
```

A2313: Nesting of include files exceeds 50

[ERROR]

Description

The maximum number of nested include files has been exceeded. The Macro Assembler supports up to 50 nested include files.

Tips

Reduce the number of nested include file to 50.

A2314: Expression must be absolute

[ERROR]

Description

An absolute expression is expected at the specified position.

1. Assembler directives expecting an absolute value are:
   - OFFSET, ORG, ALIGN, SET, BASE, DS, LLEN, PLEN, SPC, TABS, IF, IFEQ, IFNE, IFLE, IFLT, IFGE, IFGT.
2. The first operand in a DCB directive must be absolute:

Example

```
DataSec: SECTION
label1:  DS.W 1
```

```
label2:  DS.W 2
label3:  EQU  8

codeSec: SECTION

         BASE label1

         ALIGN label2
```

Tips
Specify an absolute expression at the specified position.

Example

```
DataSec: SECTION
label1:  DS.W 1
label2:  DS.W 2
label3:  EQU  8

codeSec: SECTION

         BASE label3

         ALIGN 4
```

A2316: Section name required

[ERROR]

Description
A SWITCH directive is not followed by a symbol name. Absolute expressions or string are not allowed in a SWITCH directive.
The symbol specified in a SWITCH directive must refer to a previously defined section.

Example

```
dataSec: SECTION
label1: DS.B 1

codeSec: SECTION

         SWITCH $A344
```

Tips
Specify the name of a previously define section in the SWITCH instruction.

Example

```
dataSec: SECTION
label1: DS.B 1

codeSec: SECTION

         SWITCH dataSec
```

A2317: Illegal redefinition of section name

[ERROR]

Description
The name associated with a section is previously used as a label in a code or data section or is specified in a XDEF directive.
The macro assembler does not allow to export a section name, or to use the same name for a section and a label.

Example

```
dataSec:   SECTION
sec_Label: DS.W 3
           ; ...
sec_Label: SECTION
           ; ...
```

Tips

Change to name of the section to a unique identifier.

Example

```
dataSec:   SECTION
dat_Label: DS.W 3
           ; ...
sec_Label: SECTION
           ; ...
```

A2318: Section not declared

[ERROR]

Description

The label specified in a SWITCH directive is not associated with a section.

Example

```
dataSec:   SECTION
label1:    DS.B 1
           ; ...
codeSec:   SECTION
           ; ...
           SWITCH daatSec
           ; ...
```

Tips

Specify the name of a previously define section in the SWITCH instruction.

Example

```
dataSec:   SECTION
label1:    DS.B 1
           ; ...
codeSec:   SECTION
           ; ...
           SWITCH dataSec
           ; ...
```

A2319: No section link to this label

[ERROR]

Description

A label without corresponding section was detected. This error usually occurs because of other errors before.

Tips

Correct all errors before this one first.

A2320: Value too small

[ERROR]

Description

The absolute expression specified in a directive is too small.
This message can be generated in following cases:

- The expression specified in an ALIGN, DCB or DS directive is smaller than 1.
- The expression specified in a PLEN directive is smaller than 10. A header is generated on the top of each page from the listing file. This header contains at least 6 lines. So a page length smaller than 10 lines does not make many sense.
- The expression specified in a LLEN, SPC or TABS directive is smaller than 0 (negative).

Example

```
         PLEN   5
         LLEN  -4
dataSec: SECTION
         ALIGN  0
         ; ...
label1:  DS.W   0
         ; ...
```

Tips

Modify the absolute expression to a value in the range specified above.

Example

```
         PLEN   50
         LLEN   40
dataSec: SECTION
         ALIGN  8
         ; ...
label1:  DS.W   1
         ; ...
```

A2321: Value too big

[ERROR]

Description

The absolute expression specified in a directive is too big.
This message can be generated in following cases:

- The expression specified in an ALIGN directive is bigger than 32767.
- The expression specified in a PLEN directive is bigger than 10000.
- The expression specified in a LLEN directive is bigger than 132.
- The expression specified in a SPC directive is bigger than 65.
- The expression specified in a TABS directive is bigger than 128.

Example

```
         PLEN   50000
         LLEN  200
dataSec: SECTION
         ALIGN  40000
         ; ...
```

Tips

Modify the absolute expression to a value in the range specified above.

Example

```
         PLEN   50
         LLEN   40
dataSec: SECTION
         ALIGN  8
         ; ...
```

A2323: Label is ignored

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A label is specified in front of a directive, which does not accept any label. The macro assembler ignores such label.

These labels cannot not be referenced anywhere else in the application. Labels will be ignored in front of following directives:

- ELSE, ENDIF, END, ENDM, INCLUDE, CLIST, ALIST, FAIL, LIST, MEXIT, NOLIST, NOL, OFFSET, ORG, NOPAGE, PAGE, LLEN, PLEN, SPC, TABS, TITLE, TTL.

Example

```
CodeSec: SECTION
         ; ...
label:   PLEN 50
         ; ...
label2:  LIST
         ; ...
```

Tips

Remove the label which is not required. If you need a label at that position in a section, define the label on a separate line.

Example

```
CodeSec: SECTION
         ; ...
label:
         PLEN 50
         ; ...
label2:
         LIST
         ; ...
```

A2324: Illegal Base (2,8,10,16)

[ERROR]

Description

An invalid base number follows a BASE directive. The valid base numbers are 2, 8, 10 or 16.

The expression specified in a BASE directive must be an absolute expression and must match one of the values enumerated above.

Example

```
         BASE  67
dataSec: SECTION
         ; ...
label:   EQU 35
         ; ...
         BASE  label
```

Tips

Specify one of the valid value in the BASE directive.

Example

```
         BASE  16
         ; ...
dataSec: SECTION
label:   EQU 8
         ; ...
         BASE  label
```

A2325: Comma or Line end expected

[ERROR]

Description

An incorrect syntax has been detected in a DC, FCB, FDB, FQB, XDEF, PUBLIC, GLOBAL, XREF or EXTERNAL directive.
This error message is generated when the values enumerated in one of the directive enumerated above are not terminated by an end of line character, or when they are not separated by a ',' character.

Example

```
      XDEF  dataLab1 dataLab2
      XREF  bb1, bb2, bb3, bb4    This is a comment
      ; ...
dataSec: SECTION
dataLab1: DC.B 2 | 4 | 6 | 8
dataLab2: FCB  45, 66, 88     label3:DC.B 4
```

Tips

Use the ',' character as separator between the different items in the list or insert an end of line at the end of the enumeration.

Example

```
      XDEF  dataLab1, dataLab2
      XREF  bb1, bb2, bb3, bb4    ;This is a comment
      ; ...
dataSec: SECTION
dataLab1: DC.B 2, 4, 6, 8
dataLab2: FCB  45, 66, 88
label3:   DC.B 4
```

A2326: Label <Name> is redefined

[ERROR]

Description

A label redefinition has been detected. This message is issued when:

- The label specified in front of a DS, DCB, FCC directive is already defined.
- One of the label names enumerated in a XREF directive is already defined.
- The label specified in front of an EQU directive is already defined.
- The label specified in front of a SET directive is already defined and not associated with another SET directive.
- A label with the same name as an external referenced symbol is defined in the source file

Example

```
Data1Sec:  SECTION
label1:    DS.W   4
           ; ...
Data2Sec:  SECTION
label1:    DS.W   1
           ; ...
```

Tips

Modify your source code to use unique identifiers.

Example

```
Data1Sec:  SECTION
d1_label1: DS.W   4
           ; ...
```

```
Data2Sec:  SECTION
d2_label1: DS.W   1
           ; ...
```

A2327: ON or OFF expected

[ERROR]

Description

The syntax for a MLIST or CLIST directive is not correct. These directives expects a unique operand, which can take the value ON or OFF.

Example

```
CodeSec: SECTION
         ; ...
         CLIST
         ; ...
```

Tips

Specify either ON or OFF after the MLIST or CLIST directive.

Example

```
CodeSec: SECTION
         ; ...
         CLIST ON
         ; ...
```

A2328: Value is truncated

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The size of one of the constants listed in a DC directive is bigger than the size specified in the DC directive.

Example

```
DataSec: SECTION
cst1:    DC.B  $56, $784, $FF
cst2:    DC.W  $56, $784, $FF5634
```

Tips

Reduce the value from the constant to a value fitting in the size specified in the DC directive.

Example

```
DataSec: SECTION
cst1:    DC.B  $56, $7, $84, $FF
cst2:    DC.W  $56, $784, $FF, $5634
```

A2329: FAIL found

[ERROR]

Description

The FAIL directive followed by a number smaller than 500 has been detected in the source file.

This is the normal behavior for the FAIL directive. The FAIL directive is intended for use with conditional assembly, to detect user defined error or warning condition

Example

```
LE.B:   MACRO
        IFC "\1",""
          FAIL  "no data" ; error
          MEXIT
```

```
        ENDIF
        IFC "\2",""
          FAIL  600       ; warning
          DC.B  \1
          MEXIT
        ENDIF
        IFNC "\3",""
          FAIL  400       ; error
        ENDIF
        DC.B  \2,\1
        ENDM

        LE.B    $12,$34,$56
```

A2330: String is not allowed

[ERROR]

Description

A string has been specified as initial value in a DCB directive. The initial value for a constant block can be any byte, half-word or word absolute expression as well as a simple relocatable expression.

Example

```
CstSec: SECTION
label:  DCB.B  10, "aaaaaa"
        ; ...
```

Tips

Specify the ASCII code associated with the characters in the string as initial value.

Example

```
CstSec: SECTION
label:  DCB.B  10, $61
        ; ...
```

A2332: FAIL found

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The FAIL directive followed by a number bigger than 500 has been detected in the source file.

This is the normal behavior for the FAIL directive. The FAIL directive is intended for use with conditional assembly, to detect user defined error or warning condition

Example

```
LE.B:   MACRO
        IFC "\1",""
          FAIL  "no data" ; error
          MEXIT
        ENDIF
        IFC "\2",""
          FAIL  600       ; warning
          DC.B  \1
          MEXIT
        ENDIF
        IFNC "\3",""
          FAIL  400       ; error
        ENDIF
```

```
        DC.B  \2,\1
        ENDM

        LE.B    $12
```

A2333: Forward reference not allowed

[ERROR]

Description

A forward reference has been detected in an EQU instruction. This is not allowed.

Example

```
CstSec: SECTION
        ; ...
equLab: EQU label2
        ; ...
label2: DC.W $6754
        ; ...
```

Tips

Move the EQU after the definition of the label it refers to.

Example

```
CstSec: SECTION
        ; ...
label2: DC.W $6754
        ; ...
equLab: EQU label2
        ; ...
```

A2335: Exported SET label is not supported

[ERROR]

Description

The SET directive does not allow a reference to an external label.

Example

```
        XDEF setLab
const:  SECTION
lab:    DC.B    6

setLab: SET  $77AA
```

Tips

SET labels initialized with absolute expressions can be defined in a special file to be included by assembly files, or the EQU directive can be used.

Example

```
        XDEF setLab
const:  SECTION
lab:    DC.B    6

setLab: EQU  $77AA
```

See also

SET Directive

A2336: Value too big

[DISABLED, INFORMATION, WARNING, ERROR]

Description

The absolute expression specified as initialization value for a block defined using DCB is too big. This message is generated when the initial value specified in a DCB.B directive cannot be coded on a byte.
In this case the value used to initialize the constant block will be truncated to a byte value.

Example

```
constSec: SECTION
          ; ...
label1:   DCB.B   2, 312
          ; ...
```

In the previous example, the constant block is initialized with the value $38 (= 312 & $FF)

Tips

To avoid this warning, modify the initialization value to a byte value.

Example

```
constSec: SECTION
          ; ...
label1:   DCB.B   2, 56
          ; ...
```

A2338: <FailReason>

[ERROR]

Description

The FAIL directive followed by a string has been detected in the source file.
This is the normal behavior for the FAIL directive. The FAIL directive is intended for use with conditional assembly, to detect user defined error or warning condition

Example

```
LE.B:   MACRO
        IFC "\1",""
          FAIL  "no data" ; error
          MEXIT
        ENDIF
        IFC "\2",""
          FAIL  600       ; warning
          DC.B  \1
          MEXIT
        ENDIF
        IFNC "\3",""
          FAIL  400       ; error
        ENDIF
        DC.B  \2,\1
        ENDM

        LE.B    ; no args
```

A2340: Macro parameter already defined

[ERROR]

Description

A name of a macro parameter was already defined.
Note: Not all assemblers do support named macro parameters. Assembler not supporting this will never issue this message.

A2341: Relocatable Section Not Allowed: an Absolute file is currently directly generated

[ERROR]

Description

A relocatable section has been detected while the assembler tries to generate an absolute file. This is not allowed.

Example

```
DataSec:  SECTION
Data1:    DS.W 1
          ORG $800
entry:
          NOP
          NOP
addData1: DC.W Data1
```

Tips

When you are generating an absolute file, your application should be encoded in a single source file, and should only contain absolute symbol.
So in order to avoid this message, define all your section as absolute section and remove all XREF directives from your source file.

Example

```
          ORG $1000
Data1:    DS.W 1
          ORG $800
entry:
          NOP
          NOP
addData1: DC.W Data1
```

A2342: Label in an OFFSET section cannot be exported

[ERROR]

Description

An external defined label is provided as offset in an OFFSET directive or a label defined in an offset is used in a DS directive.

Example

```
          OFFSET 1
ID:       DS.B   1
          ALIGN 4
COUNT:    DS.W   1
          ALIGN 4
VALUE:    DS.W   1
SIZE:     EQU *

          XDEF VALUE
DataSec: SECTION
Struct:  DS.B SIZE
```

Tips

Use other labels to specify the size of the offset and the number of space to provide.

Example

```
          OFFSET 1
ID:       DS.B   1
          ALIGN 4
```

```
COUNT:   DS.W   1
         ALIGN 4
VALUE:   DS.W   1
SIZE:    EQU *

DataSec: SECTION
Struct:  DS.B SIZE
```

A2345: Embedded type definition not allowed

[ERROR]

Description

The keyword STRUCT or UNION has been detected within a structured type definition. This is not allowed.
Note: Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

Example

```
myType: STRUCT
field1:   DS.W 1
field2:   DS.W 1
field3:   DS.B 1
fieldx:   STRUCT
  xx:       DS.B 1
  yy:       DS.B 1
          ENDSTRUCT
field4:   DS.B 3
field5:   DS.W 1
        ENDSTRUCT
```

Tips

Define the structured type as two separate structured types.

Example

```
typeX:  STRUCT
  xx:     DS.B 1
  yy:     DS.B 1
        ENDSTRUCT

myType: STRUCT
field1:   DS.W 1
field2:   DS.W 1
field3:   DS.B 1
fieldx:   TYPE typeX
field4:   DS.B 3
field5:   DS.W 1
        ENDSTRUCT
```

A2346: Directive or instruction not allowed in a type definition

[ERROR]

Description

An instruction or an invalid directive has been detected in a structured type definition. Only following directives are allowed in a structured type definition:

- DS, RMB, ALIGN, EVEN, LONGEVEN,
- Conditional Assembly directives (IF, ELSE, IFCC, ..)
- Directives related to the formatting of the listing file (PLEN, SPC, ...)

- XDEF, XREF, BASE

Note: Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

Example

```
myType: STRUCT
field1:   DS.W 1
field2:   DS.W 1
field3:   DS.B 1
cst:      DC.B $34
field4:   DS.B 3
field5:   DS.W 1
        ENDSTRUCT
```

Tips

Remove the invalid directive or instruction.

Example

```
myType: STRUCT
field1:   DS.W 1
field2:   DS.W 1
field3:   DS.B 1
field4:   DS.B 3
field5:   DS.W 1
        ENDSTRUCT
```

A2350: MEXIT is illegal (detected outside of a macro)

[ERROR]

Description

An MEXIT was found without a matching MACRO directive.

Example

```
        MEXIT
```

Tips

Check for the correct writing of the MACRO directive. Do not use MEXIT as label.

A2351: Expected Comma to separate macro arguments

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Macro arguments must be separated by a comma.

Example

```
constants MACRO
            DC.B \1+1, \2+1
          ENDM

          constants 1  2
```

Tips

Do not use spaces in macro parameters, instead use a comma:

```
          constants 1,2
```

A2352: Invalid Character

[ERROR]

Description

An invalid character was found during parsing.

Tips
Check the source file for binary parts

A2353: Illegal or unsupported directive SECT

[DISABLED, INFORMATION, WARNING, ERROR]

Description
The assembler did not understand the whole SECT directive. The SECT directive is only recognized when the option Option -Compat is present.

Tips
Use the SECTION directive instead.

See also
Option -Compat

A2354: Ignoring directive '<directive>'

[DISABLED, INFORMATION, WARNING, ERROR]

Description
The assembler is ignoring the specified directive.
This message is used mainly for directives which are not supported when the option Option -Compat is present.

See also
Option -Compat

A2355: Illegal size specification

[DISABLED, INFORMATION, WARNING, ERROR]

Description
The size specification given is not legal for this directive.

Tips
Use no size specification at all or use a different one.

A2356: Illegal RAD50 character

[DISABLED, INFORMATION, WARNING, ERROR]

Description
Note: Not all assemblers do support the RAD50 directive. This message is only issued by assemblers which do support the RAD50 directive.

See also
Directive RAD50

A2356: Illegal macro argument 'Argument'

[DISABLED, INFORMATION, WARNING, ERROR]

Description
Macro argument started with the [? syntax have to end with ?]. However this second pattern was not found.

See also
Macro argument grouping
Macros chapter
Option -CMacAngBrack

A2380: Cutting very long line

[DISABLED, INFORMATION, WARNING, ERROR]

Description
A line was longer than the limit 1024 characters. All remaining text is ignored.

Tips

- Split up the line into several lines.
- Remove trailing spaces and tabs.
- Use shorter identifiers.

A2381: Previous message was in this context <Context>

[DISABLED, INFORMATION, WARNING, ERROR]

Description

The previous message was in a special context. Usually this message is used to show the current macro expansion tree.

Example

```
TABLE: MACRO
     ; \1: size of table to be generated
     ; \2: current value for table
\@size: EQU \1
        if (\@size >= 2)
          TABLE \@size/2,\2
          TABLE \@size-\@size/2,\2+\@size/2
        else
          if (\@size == 1)
            DC \2
          endif
        endif
       ENDM

            TABLE 4
```

Generates the following messages:

```
D:\test\b.asm(9): ERROR A1055: Error in expression
INFORMATION Macro Expansion             DC

b.asm(5): INFORMATION A2381: Previous message was in this context
'Macro Invocation'
b.asm(5): INFORMATION A2381: Previous message was in this context
'Macro Invocation'
b.asm(14): INFORMATION A2381: Previous message was in this
context 'Macro Invocation'
```

So the error happens at line 9 (“DC \2”) which was called by line 5 twice and finally by line 14.
To fix this example, add a second parameter to the TABLE macro call:

```
            TABLE 4,0
```

Tips

Check the message before the first A2381 to see the cause of the problem.

A2382: Illegal character ('\0') in source file

[ERROR]

Description

An zero byte (a byte with ASCII code 0) was found in the source.

Tips

Check if the source file is binary.

A2383: Input line too long

[ERROR]

Description

An input line is longer then the translation limit.
Input lines must not be longer than 1024 characters.

Tips

Split the input line.
In recursive macros, use local SET labels to avoid lines growing with the input buffer:
Instead of:

```
TableTo: MACRO
           if (\1 > 0)
             DC.W \1
             TableTo \1 - 1
           endif
         ENDM
```

Use:

```
TableTo: MACRO
           if (\1 > 0)
             DC.W \1
             \@LocLabel: SET \1-1
             TableTo \@LocLabel
           endif
         ENDM
```

A2400: End of Line expected

[DISABLED, INFORMATION, WARNING, ERROR]

Description

The assembler did not expect anything anymore on a line. This message can be generated when:

- A comment, which does not start with the start of comment character (';'), is specified after the instruction.
- A further operand is specified in the instruction.
- ...

Example

```
DataSec: SECTION
var:     DS.B 1     Char variable
```

Tips

Remove the invalid character or sequence of characters from the line.

- Insert the start of comment character at the beginning of the comment.
- Remove the superfluous operand.
- ...

Example

```
DataSec: SECTION
var:     DS.B 1     ; Char variable
```

A2401: Complex relocatable expression not supported

[ERROR]

Description

A complex relocatable expression has been detected. A complex relocatable expression is detected when the expression contains:

- An operation between labels located in two different sections.
- A multiplication, division or modulo operation between two labels.

- The addition of two labels located in the same section.

Example

```
DataSec1: SECTION SHORT
DataLbl1: DS.B 10
DataSec2: SECTION SHORT
DataLbl2: DS.W 15
offset:   EQU DataLbl2 - DataLbl1
```

Tips

The macro assembler does not support complex relocatable expressions. The corresponding expression must be evaluated at execution time.

Example

```
DataSec1: SECTION SHORT
DataLbl1: DS.B 10
DataSec2: SECTION SHORT
DataLbl2: DS.W 15
Offset:   DS.W 1
…
CodeSec: SECTION
  …
evalOffset:
  LDD  #DataLbl2
  SUBD  #DataLbl1
  STD  Offset
```

If both `DataSec1 and DataSec2` are in the same section and defined in this module, the assembler can compute the difference:

```
DataSec1: SECTION SHORT
DataLbl1: DS.B 10
DataLbl2: DS.W 15
offset:   EQU DataLbl2 - DataLbl1
```

A2402: Comma expected

[ERROR]

Description

A comma character is missing between two operands of an instruction or directive.

Example

```
DataSec:  SECTION
Data:     DS.B  1
ConstSec: SECTION
          DC.B  2 3
```

Tips

The comma (‘,’) character is used as separator between instruction operands.

Example

```
DataSec:  SECTION
Data:     DS.B  1
ConstSec: SECTION
          DC.B  2, 3
```

A2500: Equal expected

[ERROR]

Description
In a for directive, a = was expected.

Example
```
  FOR j := $1000 TO $1003
   DC.W j
  ENDFOR
```
Tips:
Just use an equal in the example (no colon).
```
  FOR j = $1000 TO $1003
   DC.W j
  ENDFOR
```
Check that the Option -Compat=b is enabled.

See also
Option -Compat
Directive FOR

A2501: TO expected

[ERROR]

Description
In a for directive, a TO was expected.

Example
```
  FOR j := $1000 < $1003
   DC.W j
  ENDFOR
```
Tips:
Just use a TO in the example.
```
  FOR j = $1000 TO $1003
   DC.W j
  ENDFOR
```
Check that the Option -Compat=b is enabled.

See also
Option -Compat
Directive FOR

A2502: ENDFOR missing

[ERROR]

Description
In a for directive, a TO was expected.

Example
```
  FOR j := $1000 < $1003
   DC.W j
```
Tips:
Check that every FOR has a corresponding ENDFOR.
```
  FOR j = $1000 TO $1003
   DC.W j
  ENDFOR
```
Check that the Option -Compat=b is enabled.

See also
Option -Compat
Directive FOR

A2503: ENDFOR without FOR

[ERROR]

Description

A ENDFOR without corresponding FOR was found.

Example

```
;  FOR j := $1000 < $1003
   DC.W j
  ENDFOR
```

Tips:

Check that every ENDFOR has a corresponding FOR. In the example, remove the semicolon.

```
  FOR j = $1000 TO $1003
   DC.W j
  ENDFOR
```

Check that the Option -Compat=b is enabled.

See also

Option -Compat
Directive FOR

A3000: User requested stop

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is used when the user presses the stop button in the graphical user interface.
Also when the assembler is closed during an assembly, this message is issued.

Tips

By moving this message to a warning or less, the stop functionality can be disabled.

A4000: Recursive definition of label <Label name>

[ERROR]

Description

The definition of an EQU label depends directly or indirectly on itself.

Example

```
        XDEF  tigger
pooh:   EQU tigger - 2
tigger: EQU 2*pooh
```

Tips

This error usually indicates an error in some definitions. Determine the labels involved in the recursive definition and eliminate the circular dependency.

A4001: Data directive contains no data

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A data directive is empty, and no code is generated for this directive.

Example

```
        DC.B    ; 1,2,3,4
```

Tips

This warning may indicate an error, or it may be intentional within a macro expansion, for example.

A4002: Variable access size differs from previous declaration

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An implicit or explicit declaration of a label indicates an access size which differs from a former declaration.

Tips

Indicating the access size of variables is particularly helpful in “header” files which contain XREF directives, to be included by other files accessing these variables. If an assembly file contains a “XREF.B obj”, and the header file declares “XREF.W obj”, this warning message indicates potential problems.

A4003: Found XREF, but no XDEF for label <Label>, ignoring XREF

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The local definition of a label <Label> supersedes a global XREF declaration, if no appropriate XDEF directive is given.

Example

```
        XREF  main
Code:   SECTION
main:   NOP       ; is local, unless XDEF given
        NOP
```

Tips

This warning may indicate a forgotten “XDEF” directive.

A4004: Qualifier ignored

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An unknown qualifier to a SECTION or ORG directive is ignored.

Example

```
const:  SECTION SHORT 1234 FOO
        DC.B    "hello", 0
```

Tips

This warning may indicate a misspelled qualifier.

A4005: Access size mismatch for <Symbol>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Incompatible access sizes are attached to an object, either implicitly or explicitly. The access size of an object is determined from XREF declarations, XDEF definitions and (if applicable) from the access size of the section, where the object is placed into.

Example

```
        XDEF.B  two
const:  SECTION
two:    DC.B    2   ; implicit *.W definition
```

Tips

It is probably a good idea to eliminate mismatches, particularly if mismatches occur between declarations in a “header file” and definitions in the assembly file.

A4100: Address space clash for <Symbol>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is only relevant for Harvard architectures (separate code and data address spaces), and occurs for symbols whose address is used both as a code address and a data address.

Tips

This clash may be intentional, but indicates an error in most cases.

A12001: Illegal Addressing Mode

[ERROR]

Description

An illegal addressing mode has been detected in an instruction. This message is generated when an incorrect encoding is used for an addressing mode.

Example

```
LDD [D X]
LDD [D, X
ANDCC $FA
```

Tips

Use a valid notation for the addressing mode encoding.

Example:

```
LDD [D, X]
ANDCC #$FA
```

A12003: Value is truncated to one byte

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A word operand is specified in an assembly instruction expecting a byte operand. This warning may be generated in following cases:

- •1. A symbol located in a section, which is accessible using the extended addressing mode, is specified as operand in an instruction expecting a direct operand.
- •2. An external symbol imported using XREF is specified as operand in an instruction expecting a direct operand.
- •3. The mask specified in a BCLR, BSET, BRCLR or BRSET is bigger than 0xFF.

Example

```
          XREF extData
dataSec: SECTION
data:    DS.B 1
data2:   DS.B 1
destination: DS.W 1
codeSec: SECTION
          MOVB    #data, destination
          MOVB    #data, destination
          MOVB    #extData, destination
          BCLR    data, #$54F
```

Tips

According to the reason why the warning was generated, the warning can be avoided in the following way:

- •1. Specify the force operator .B at the end of the operand or < in front of the operand.
- •2. User XREF.B to import the symbol.

Example:

```
          XREF.B extData
dataSec: SECTION
data:    DS.B 1
data2:   DS.B 1
destination: DS.W 1
codeSec: SECTION
```

```
MOVB    #data.B, destination
MOVB    #extData, destination
BCLR    data, #$4F
```

A12004: Value is truncated to two bytes

[DISABLE, INFORMATION, WARNING, ERROR]

Description

If a value is larger than two bytes, but the instruction only allows a 16bit value, this message is issued.

A12005: Value must be between 1 and 8

[ERROR]

Description

The expression specified in a pre increment, post increment, pre decrement or post decrement addressing mode is out of the range [1..8]

Example

```
STX   10, SP+
```

Tips

According to the HC12 addressing mode notation, the increment or decrement factor must be bigger than 0 and smaller than 9.

A12006: Value is truncated to five bits

[DISABLE, INFORMATION, WARNING, ERROR]

Description

If a value is larger than five bits, but the instruction only allows a 5bit value, this message is generated.

A12008: Relative branch with illegal target

[ERROR]

Description

The offset specified in a PC relative addressing mode is a complex relocatable expression, a symbol defined in another section or an external defined symbol.

Example

```
DataSec: SECTION
Data:    DS.B  1
Code1Sec: SECTION
Entry1:
  NOP
  LDD #$6000
  STD Data
CodeSec: SECTION
  LDD   Data
  CPD   #$6000
  BNE   Entry1
```

A12009: Illegal expression

[ERROR]

Description

An illegal expression is specified in a PC relative addressing mode. The illegal expression may be generated in following cases:

- 1. A complex expression is specified, when a PC relative expression is expected.
- 2. A left or right parenthesis is missing in the expression.

Example

```
CodeSec1: SECTION
Entry1:
  NOP
CodeSec2: SECTION
Entry2:
  NOP
  BRA Entry2 - Entry1

  BRA (Entry2 + 1
```

Tips
Change the expression to a valid expression.

Example:

```
CodeSec1: SECTION
Entry1:
  NOP
CodeSec2: SECTION
Entry2:
  NOP
  BRA Entry2

  BRA (Entry2 + 1)
```

A12010: Register expected

[ERROR]

Description
A register mnemonic is missing in a post increment, post decrement, pre increment or pre decrement addressing mode.

Example

```
LDD   1, -ssp
```

Tips
Specify a register mnemonic on the specified position.

Example

```
LDD   1, -sp
```

A12102: Page value expected

[ERROR]

Description
A page number is missing in a CALL instruction.

Example

```
DataSec: SECTION
data: DS.L 2
FarCodeSec: SECTION
FarFunction:
    LDD   #45
    STD   data
CodeSec:    SECTION
…
  CALL  FarFunction
```

Tips
Add the missing page operand to the CALL instruction

Example:

```
DataSec: SECTION
data: DS.L 2
FarCodeSec: SECTION
FarFunction:
    LDD   #45
    STD   data
CodeSec:    SECTION
…
  CALL  FarFunction, PAGE(FarFunction)
```

A12103: Operand not allowed

[ERROR]

Description

The operand specified in an assembly instruction is not valid for this instruction.

Example

```
DataSec: SECTION
data DS.B 20
…
CodeSec: SECTION
LEAX #data
```

Tips

Check your HC12 User's Guide and modify the source code in order to have only valid instructions and addressing mode combination.

Example:

```
DataSec: SECTION
data DS.B 20
…
CodeSec: SECTION
LDX #data
```

A12104: Immediate value expected

[ERROR]

Description

The immediate addressing mode is expected at that position. Usually this error message is generated when the mask specified in a BRCLR or BRSET instruction is not preceded by the immediate character ('#').

Example

```
maskValue: EQU $40
         BSCT
var:     DS.B 1
CodeSec: SECTION
entry:
         LDD   #4567
         BRCLR var, maskValue, endCode
         …
endCode:
         END
```

Tips

Insert the character '#' at the requested position to change to the immediate addressing mode.

Example:

```
maskValue: EQU $40
         BSCT
var:     DS.B 1
CodeSec: SECTION
entry:
         LDD   #4567
         BRCLR var, #maskValue, endCode
         …
endCode:
         END
```

A12105: Immediate Address Mode not allowed

[ERROR]

Description

The immediate addressing mode is not allowed at that position. Usually this message is generated when the first operand specified in a BCLR, BSET, BRCLR or BRSET instruction is preceded by the immediate character ('#').

Example

```
maskValue: EQU $40
         BSCT
var:     DS.B 1
CodeSec: SECTION
entry:
         LDD   #4567
         BRCLR #var, #maskValue, endCode
         …
endCode:
         END
```

Tips

Remove the unexpected '#' character.

Example:

```
maskValue: EQU $40
         BSCT
var:     DS.B 1
CodeSec: SECTION
entry:
         LDD   #4567
         BRCLR var, #maskValue, endCode
         …
endCode:
         END
```

A12107: Illegal size specification for HC12-instruction

[ERROR]

Description

A size operator follows an HC12 instruction. Size operators are coded as semicolon character followed by single character.

Example

```
CodeSec: SECTION
  …
  ADDD.W #$0076
```

Tips

Remove the size specification following the HC12 instruction.

Example:

```
CodeSec: SECTION
  …
  ADDD #$0076
```

A12111: Invalid Offset in TRAP instruction. valid offsets are $30 .. $39 and $40 .. $FF

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An illegal offset has been specified in a TRAP instruction. The offset has to be either in the range from 0x30 to 0x39 or in 0x40 to 0xff.

A12202: Not a hc12 instruction or directive

[ERROR]

Description

The identifier detected in an assembly line instruction part is neither an assembly directive, nor an HC12 instruction, nor a user defined macro.

Example

```
CodeSec: SECTION
  …
   LDHX #$5510
```

Tips

Change the identifier to a valid assembly directive, to a HC12 instruction or to the name of a user defined macro.

Example:

```
CodeSec: SECTION
  …
   LDD #$5510
```

A12403: Value out of range -256..255

[ERROR]

Description

The offset between the current PC and the label specified as PC relative address is not in the range of a signed 9–bits value (smaller than -256 or bigger than 255). A 9 bit signed PC relative offset is expected in following instructions:

- Decrement and-branch instructions
  DBEQ, DBNE
- Increment and-branch instructions
  IBEQ, INE
- Test and-branch instructions
  TBEQ, TBNE

Example

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 10
CodeSec: SECTION
         …
         LDX  #var2
label:   LDD  var1
         CLR  1, X+
```

```
dummyBl: DCB.B 260, $A7
         DBNE D, label
```

Tips

Replace the instruction by the following portion of code:

•For decrement and branch:

Ifcc	Condition
`DBNE D, label`	`SUBD #1` `LBNE label`
`DBNE A, label`	`DECA` `LBNE label`
`DBNE B, label`	`DECB` `LBNE label`
`DBNE X, label`	`DEX` `LBNE label`
`DBNE Y, label`	`DEY` `LBNE label`
`DBNE S, label`	`DES` `LBNE label`

•For increment and branch:

Ifcc	Condition
`IBNE D, label`	`ADDD #1` `LBNE label`
`IBNE A, label`	`INCA` `LBNE label`
`IBNE B, label`	`INCB` `LBNE label`
`IBNE X, label`	`INX` `LBNE label`
`IBNE Y, label`	`INY` `LBNE label`
`IBNE S, label`	`INS` `LBNE label`

•For test and branch:

Ifcc	Condition
`TBNE D, label`	`CPD #0` `LBNE label`
`TBNE A, label`	`TSTA` `LBNE label`
`TBNE B, label`	`TSTB` `LBNE label`
`TBNE X, label`	`CPX #0` `LBNE label`
`TBNE Y, label`	`CPY #0` `LBNE label`
`TBNE S, label`	`CPS #0` `LBNE label`

Example:

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 10
CodeSec: SECTION
         …
         LDX  #var2
label:   LDD  var1
         CLR  1, X+
dummyBl: DCB.B 260, $A7
         SUBD #1
         LBNE label
```

A12404: Value out of range -16..15

[ERROR]

Description

The offset used does not fit into the instruction addressing mode range between -16 and 15.

A12409: In PC relative addressing mode, references to object located in another section or file are only allowed for IDX2 addressing mode.

[ERROR]

Description

An reference to an external symbol or a symbol defined in another section is detected in an 9- bits or 5-bits indexed PC relative addressing mode. This is not allowed.

Example

```
dataSec:   SECTION
data:      DS.W 1
cstSec:    SECTION
label:     DC.W $33A5, $44BA
```

```
codeSec1:  SECTION
entry:
           MOVB  label, PCR, data
```

Tips

Merge the sections containing the symbol and the instruction together or change the instruction to an instruction supporting the 16-bit indexed PC relative addressing mode.

Example of Merging sections:

```
dataSec:   SECTION
data:      DS.W 1
codeSec1:  SECTION
label:     DC.W $33A5, $44BA
entry:
           MOVB  label, PCR, data
```

Example of Changing Instruction:

```
dataSec:   SECTION
data:      DS.W 1
cstSec:    SECTION
label:     DC.W $33A5, $44BA
codeSec1:  SECTION
entry:
           LDD   label, PCR
           STD   data
```

A12411: Restriction: label specified in a DBNE, DBEQ, IBNE, IBEQ, TBNE or TBEQ instruction should be defined in the same section they are used.

[ERROR]

Description

An external symbol or a symbol defined in another section has been detected in a **DBNE, DBEQ, IBNE, IBEQ, TBNE or TBEQ** instruction.
This is not allowed in a relocatable section.

Example

```
dataSec:   SECTION
data:      DS.W 1
codeSec0:  SECTION
label:
           NOP
           NOP
codeSec1:  SECTION
entry:
           DBNE  A, label
```

Tips

Merge the sections containing the symbol and the instruction together or change the instruction to an instruction supporting the 16-bit indexed PC relative addressing mode.

Example of Merging sections:

```
dataSec:   SECTION
data:      DS.W 1
codeSec0:  SECTION
label:
           NOP
```

```
            NOP

entry:
          DBNE  A, label
```

Example of Changing Instruction:

```
dataSec:   SECTION
data:      DS.W 1
codeSec0:  SECTION
label:
           NOP
           NOP
codeSec1:  SECTION
entry:
          DECA
          BNE   label
```

A12412: PCR is ignored for this addressing mode

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The PCR keyword is treated like a PC keyword for the Accumulator D Indirect Indexed Addressing mode. The PCR keyword does differ from the PC register keyword only in the way how the offset is encoded. This addressing mode has no fix offset, and therefore PC and PCR behave identical.

Example

```
LDAA [D,PCR]
```

A12600: Address lower than segment current position

[ERROR]

Description

A location is smaller than the last location used in the segment.

A12704: DEFSEG is missing

[ERROR]

Description

In avocet compatibility mode, a name after a SEG directive was not defined with a DEFSEG directive.

Example

```
  DEFSEG MyCode CODE
  DEFSEG MyData DATA
  nop
  SEG MyCodeData
  nop
  nop
```

Tips:

Check the spelling.

```
  DEFSEG MyCode CODE
  DEFSEG MyData DATA
  nop
  SEG MyData
  nop
  nop
```

```
                XREF label

                ASR label
                BCLR 2,label
                BRSET 0, label, *
                BSET 0,label
                CBEQ label,*
                CLR label
                COM label
                CPHX label; legal for HCS08
                DBNZ label, *
                DEC label
                INC label
                LDHX label; legal for HCS08
                LSL label
                LSR label
                MOV #1, label
                MOV label, label
                NEG label
                ROL label
                ROR label
                STHX label; legal for HCS08
                TST label


        ; Example2
                XREF.B label

                ASR label
                BCLR 2,label
```

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X