MA039–000–00–00
Doc. ver.: 1.22

DSP56xxx v3.6

CROSS–ASSEMBLER, LINKER/LOCATOR, UTILITIES USER'S GUIDE

TASKING

A publication of

Altium BV

Documentation Department

http://www.tasking.com
http://www.altium.com

CONTENTS

TABLE OF CONTENTS

TASKING

CONTENTS

MIGRATION FROM MOTOROLA CLAS F-1

DESCRIPTIVE LANGUAGE FOR EMBEDDED ENVIRONMENTS G-1

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the DSP5600x, DSP563xx and DSP566xx cross–assembler, linker, locator and utilities. It assumes that you are familiar with programming the DSP5600x, DSP563xx or DSP566xx.

INSTALLATION

The software installation is described in the *C Cross–Compiler User's Guide*.

MANUAL STRUCTURE

Related Publications

Conventions Used In This Manual

1. Overview
   Makes you familiar with the assembler itself, through the use of sample programs.

2. Assembler
   Describes the actions and invocation of the TASKING DSP56xxx assemblers (**as56** or **as563**).

3. Software Concept
   Describes the basics of modular programming and sections.

4. Assembly Language
   Describes the formats of the possible statements for an assembly program.

5. Operands and Expressions
   Describes the operands and expressions to be used in the assembler instructions and directives.

6. Macro Operations
   Describes the use of macros and conditional assembly.

7. Assembler Directives
   Describes the assembler directives to pass information to the assembler program.

8. Structured Control Statements
   Describes the use of structured control statements for loops and conditional branches.

9. Instruction Set
   Gives a list of assembly language instruction mnemonics.

10. Linker
    Describes the action of, and options/controls applicable, to the linker.

11. Locator
    Describes the action of, and options/controls applicable, to the locator.

12. Utilities
    Contains descriptions of the utilities supplied with the package, which may be useful during program development.

APPENDICES

A. Assembler Error Messages
   Gives a list of error messages which can be generated by the assembler.

B. Linker Error Messages
   Gives a list of error messages which can be generated by the linker.

C. Locator Error Messages
   Gives a list of error messages which can be generated by the locator.

D. Archiver Error Messages
   Gives a list of error messages which can be generated by the archiver.

E. Embedded Environment Error Messages
   Gives a list of error messages from the embedded environment which can be generated by the linker/locator.

F. Migration from Motorola CLAS
   Describes how you can migrate your assembly program from the Motorola CLAS assembler to one of the TASKING DSP56xxx assemblers (**as56** or **as563**).

G. DEscriptive Language For Embedded Environments
Describes the Delfee description language.

H. Delfee Syntax
Contains a syntax description of the Delfee language.

I. IEEE–695 Object Format
Contains a description of the IEEE–695 object format and the TIOF format.

J. Motorola S–Records
Contains a description of the Motorola S–records.

K. Intel Hex Records
Contains a description of the Intel Hex format.

INDEX

RELATED PUBLICATIONS

TASKING Tools

- DSP56xxx C Cross–Compiler User's Guide
  [TASKING, MA039–002–00–00]
- DSP56xxx CrossView Pro Debugger User's Guide
  [TASKING, MA039–049–00–00]

Core Reference Manuals

- DSP56000 Digital Signal Processor Family Manual [Motorola, Inc.]
- DSP560*xx* Digital Signal Processor User's Manual [Motorola, Inc.]

- DSP56300 24–Bit Digital Signal Processor Family Manual
  [Motorola, Inc.]
- DSP563*xx* 24–Bit Digital Signal Processor User's Manual
  [Motorola, Inc.]
- DSP56L307 24–Bit Digital Signal Processor User's Manual
  [Motorola, Inc.]

- DSP56600 Digital Signal Processor Family Manual [Motorola, Inc.]
- DSP5660*x* Digital Signal Processor User's Manual [Motorola, Inc.]

- DSP56652 Baseband Digital Signal Processor User's Manual
  [Motorola, Inc.]
- DSP56654 Baseband Digital Signal Processor User's Manual
  [Motorola, Inc.]

CONVENTIONS USED IN THIS MANUAL

The notation used to describe the format of call lines is given below:

{ }	Items shown inside curly braces enclose a list from which you must choose an item.
[ ]	Items shown inside square brackets enclose items that are optional.
|	The vertical bar separates items in a list. It can be read as OR.
italics	Items shown in italic letters mean that you have to substitute the item. If italic items are inside square brackets, they are optional. For example: *filename* means: type the name of your file in place of the word *filename*.
...	An ellipsis indicates that you can repeat the preceding item zero or more times.
`screen font`	Represents input examples and screen output examples.
bold font	Represents a command name, an option or a complete command line which you can enter.

For example

```
command [option]... filename
```

This line could be written in plain English as: execute the command *command* with the optional options *option* and with the file *filename*.

Illustrations

The following illustrations are used in this manual:

📌 This is a note. It gives you extra information.

⚠ This is a warning. Read the information carefully.

This illustration indicates actions you can perform with the mouse.

This illustration indicates keyboard input.

This illustration can be read as "See also". It contains a reference to another command, option or section.

CHAPTER

1

OVERVIEW

TASKING

CHAPTER 1

1.1 INTRODUCTION

TASKING offers a complete toolchain for the Motorola DSP56xxx Family of Digital Signal Processors (DSPs) and their derivatives. The DSP5600x (24–bit), the DSP563xx (24–bit) and the DSP566xx (16–bit) versions of the DSP56xxx family are supported. This manual uses 'DSP5600x' to indicate the derivatives that have a '0' in the third and fourth position (e.g. DSP56002) and 'DSP563xx' and 'DSP566xx' along the same lines. In this manual all core versions are treated identical unless implementation differences require otherwise. 'DSP56xxx' is used as a shorthand notation for the Motorola DSP56xxx Family of Digital Signal Processors (DSPs) and their derivatives.

The DSP5600x and DSP563xx/DSP566xx family toolchain produces load files running on the DSP5600x, and DSP563xx/DSP566xx family respectively. The assembler **as56** accepts programs written according to the Motorola assembly language specification for the DSP5600x and **as563** accepts programs written according to the Motorola assembly language specification for the DSP563xx and DSP566xx. The assemblers are compatible with the Motorola CLAS assembler for the DSP56xxx family. However, there are some implementation differences. For more information see Appendix F, *Migration from Motorola CLAS*.

The assembler generates relocatable object files in the IEEE–695 object format. This file format specifies code parts as well as symbol definition and symbolic debug information parts. The locator optionally produces absolute output files in Motorola S–record format or Intel Hex format. You can load these formats into a PROM programmer. The locator is also capable of generating Motorola CLAS COFF object format files. You can use this format to load a program in the Motorola CLAS simulator.

The DSP563xx/DSP566xx toolchain contains the following programs (for the DSP5600x toolchain the executable names end in '**56**'):

cc563	A handy control program which activates the other programs depending on its input files.
c563	The C compiler which produces an assembly file.
as563	The assembler program which produces a relocatable object file from a given assembly file.
lk563	A linker that links several objects and object libraries into one target load file.

lc563	A locator that links one or more linker output files into one absolute load file. This program can also produce files in Motorola S–record format, Intel Hex format and Motorola CLAS COFF object format.
ar563	An IEEE archiver. This is a librarian facility, which can be used to create and maintain object libraries.
pr563	An IEEE object reader. This utility dumps the contents of IEEE files which have been created by a tool from the TASKING DSP563xx/DSP566xx family toolchain.
mk563	A utility program to maintain, update and reconstruct groups of programs.

All DSP563xx/DSP566xx executables are used in this manual as the general names for both DSP56xxx toolchains, unless explicitly stated otherwise.

1.2 DSP56XXX FAMILY PROGRAM DEVELOPMENT

The DSP56xxx family toolchain provides an environment for modular program development and debugging. The following diagram shows the structure of the toolchain.

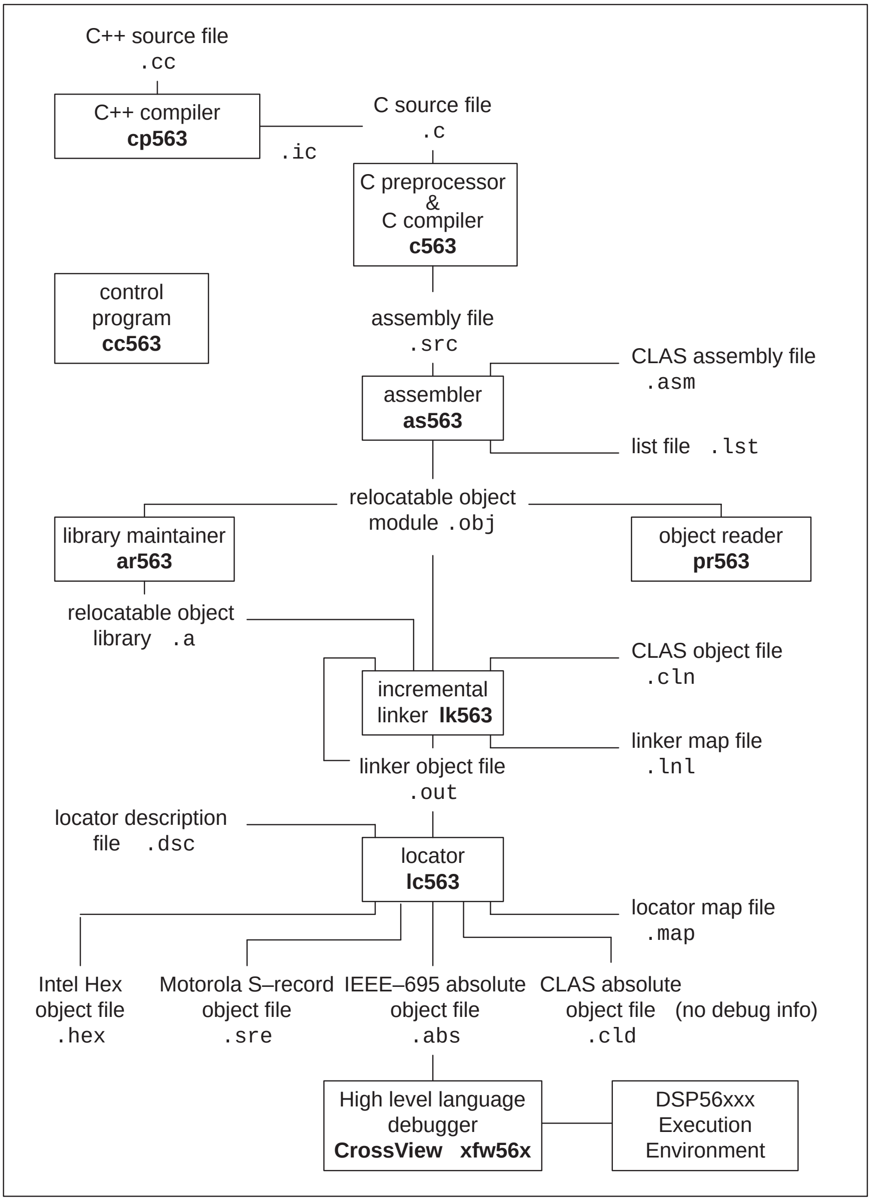


Figure 1–1: DSP563xx development flow

1.3 DEFINITION OF TERMS

Since the Motorola DSP architectures are different from normal microprocessors, the programmer may not be familiar with some of the terms used in this document. The following discussion serves to clarify some of the concepts discussed later in this manual.

The Motorola DSP5600x, DSP563xx and DSP566xx architecture can have as many as five separate memory spaces referred to as the **X, Y, L, P** (**P**rogram), and **E** (EMI – Extended Memory Interface) memory spaces. **L** memory space is a concatenation of **X** and **Y** data memory and is considered by the assembler as a superset of the **X** and **Y** memory spaces. **E** memory is specific to the DSP56004 processor, and provides for different data representations for various memory hardware configurations. The assembler can generate object code for each memory space, but object code is restricted to one memory space at a time.

The Motorola digital signal processors are capable of performing operations on modulo and reverse–carry **buffers**, two data structures useful in digital signal processing applications. The assembler provides directives for establishing buffer base addresses, allocating buffer space, and initializing buffer contents. For a buffer to be located properly in memory the lower bits of the starting address which encompass the buffer size must be zero. For example, the lowest address greater than zero at which a buffer of size 32 may be located is 32 (20 hexadecimal). More generally, the buffer base address must be a multiple of 2^k, where 2^k is greater than or equal to the size of the buffer. Buffers can be allocated manually or by using the assembler buffer directives (see Chapter 7, *Assembler Directives*).

1.4 BASIC ASSEMBLY, LINKING AND LOCATING OF A DSP56XXX PROGRAM

This section illustrates the typical input format of a DSP56xxx assembly program for the cross–assembler. As a part of the installation a directory `examples\asm` (`examples/asm` for UNIX) is created depending on the place where you installed the package on your system. This example directory contains, among others, the following assembly source files:

startup.asm calc.asm

Note that this program has been written for illustrative purposes only.

1.4.1 USING EDE

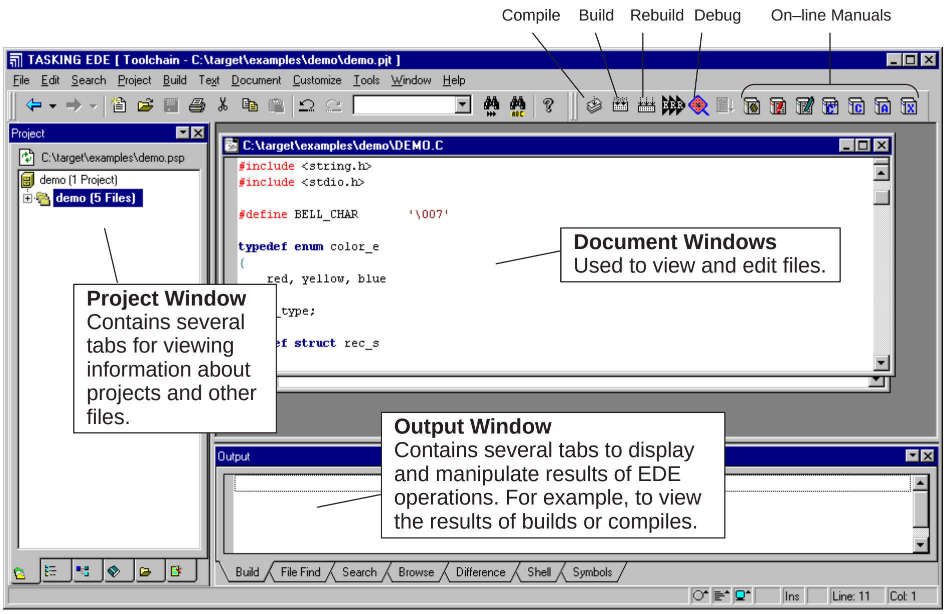
EDE stands for ”Embedded Development Environment” and is the Windows oriented Integrated Development Environment you can use with your TASKING toolchain to design and develop your application.

To use EDE on the `calc` demo program in the subdirectory `asm` in the `examples` subdirectory of the DSP56xxx product, follow the steps below.

How to Start EDE

You can launch EDE by double–clicking on the EDE shortcut on your desktop.

The EDE screen provides you with a menu bar, a toolbar (command buttons) and one or more windows (for example, for source files), a status bar and numerous dialog boxes.


Compile
Build
Rebuild
Debug
On–line Manuals
TASKING EDE [Toolchain - C:\target\examples\demo\demo.pjt]
File Edit Search Project Build Text Document Customize Tools Window Help
Project
C:\target\examples\demo.psp
demo (1 Project)
demo (5 Files)
C:\target\examples\DEMO.C
Output
Build
File Find
Search
Browse
Difference
Shell
Symbols
Ins
Line: 11
Col: 1

```
#include <string.h>
#include <stdio.h>

#define BELL_CHAR       '\007'

typedef enum color_e
{
    red, yellow, blue

 _type;

 ef struct rec_s
```

Document Windows
Used to view and edit files.

Project Window
Contains several tabs for viewing information about projects and other files.

Output Window
Contains several tabs to display and manipulate results of EDE operations. For example, to view the results of builds or compiles.

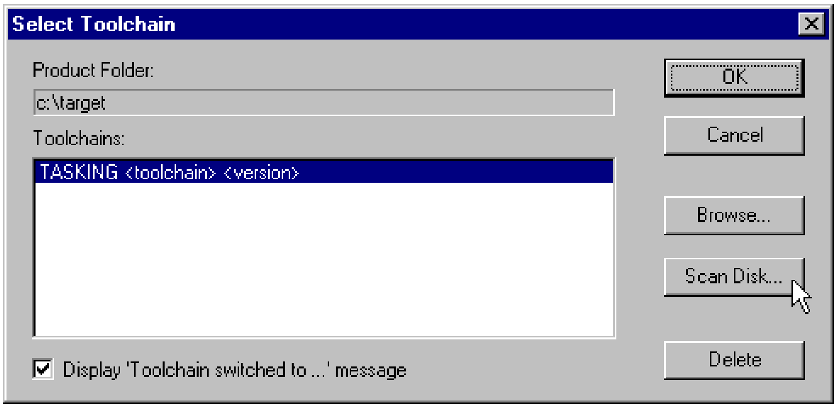
How to Select a Toolchain

EDE supports all the TASKING toolchains. When you first start EDE, the correct toolchain of the product you purchased is selected and displayed in the title of the EDE desktop window.

If you have more than one TASKING product installed and you want to change toolchains, do the following::

1. From the `Project` menu, select `Select Toolchain...`

   The `Select Toolchain` dialog appears.



2. Select the toolchain you want. You can do this by clicking on a toolchain in the `Toolchains` list box and click `OK`.

   If no toolchains are present, use the `Browse...` or `Scan Disk...` button to search for a toolchain directory. Use the `Browse...` button if you know the installation directory of another TASKING product. Use the `Scan Disk...` button to search for all TASKING products present on a specific drive. Then return to step 2.

How to Open an Existing Project

Follow these steps to open an existing project:

1. Access the `Project` menu and select `Set Current...`.

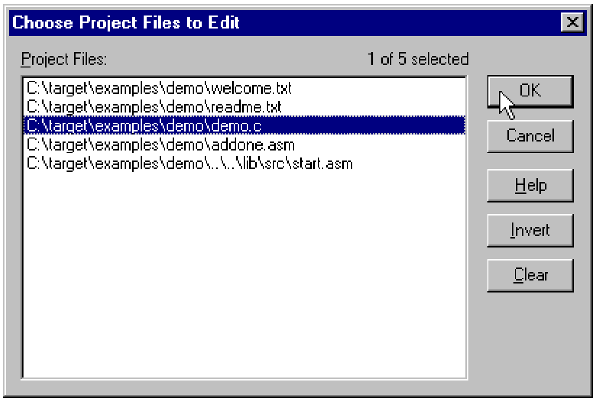
2. Select the project file to open. For the `calc` demo program select the file `asm.pjt`, located in the subdirectory `asm` in the `examples` subdirectory of the DSP56xxx product tree. If you have used the defaults, the file `asm.pjt` is in the directory `c:\c563\examples\asm` for the DSP53xx/DSP566xx (use `c56` for the DSP5600x).

How to Load/Open Files

The next two steps are not needed for the demo program because the files `sieve.c` and `makefile` are already open. To load the file you want to look at.

1. From the `Project` menu, select `Load Files...`

   The `Choose Project Files to Edit` dialog appears.


Choose Project Files to Edit

Project Files: 1 of 5 selected

C:\target\examples\demo\welcome.txt
C:\target\examples\demo\readme.txt
C:\target\examples\demo\demo.c
C:\target\examples\demo\addone.asm
C:\target\examples\demo\..\..\lib\src\start.asm

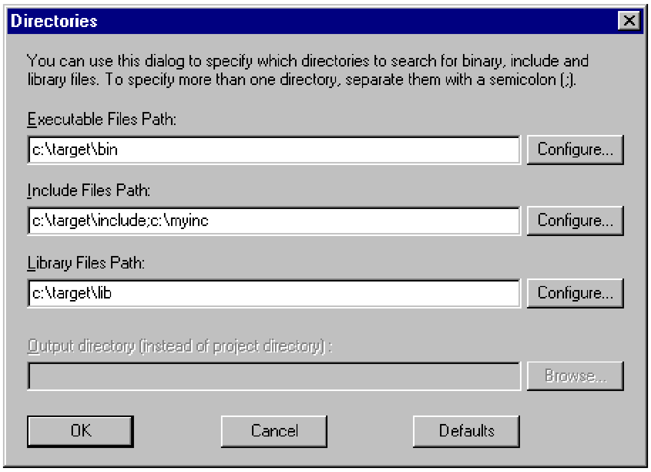
2. Choose the file(s) you want to open by clicking on it. You can select multiple files by pressing the `<Ctrl>` or `<Shift>` key while you click on a file. With the `<Ctrl>` key you can make single selections and with the `<Shift>` key you can select everything from the first selected file to the file you click on. Then click `OK`.

   This launches the file(s) so you can edit it (them).

Check the directory paths

1. From the `Project` menu, select `Directories...`.

   The `Directories` dialog appears.


Directories

You can use this dialog to specify which directories to search for binary, include and library files. To specify more than one directory, separate them with a semicolon (;).

Executable Files Path: c:\target\bin

Include Files Path: c:\target\include;c:\myinc

Library Files Path: c:\target\lib

Output directory (instead of project directory) :

2. Check the directory paths for programs, include files and libraries. You can add your own directories here, separated by semicolons.

3. Click `OK`.

How to Select a CPU Type

The next step is to compile the file(s) together with its dependent files so you can debug the application. But first you need to specify for which CPU type you want to build your application:

1. From the `Project` menu, select `Project Options...`

   The Project Options dialog box appears.

2. Select `CPU Selection`.

3. In the `CPU family/type` box, select the CPU or CPU family for which you want to build your application and click `OK`.

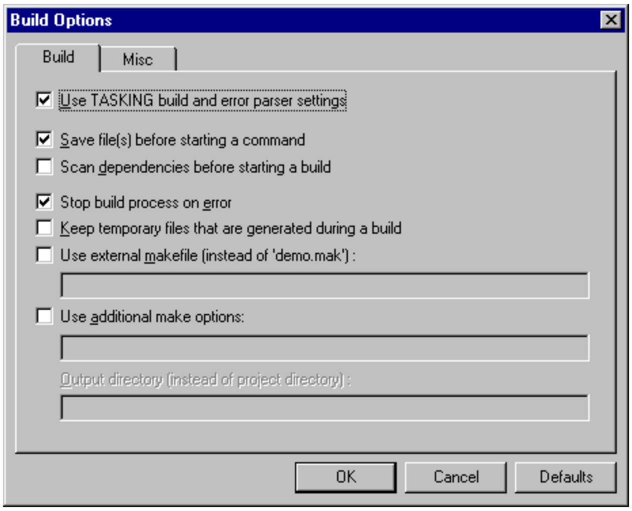
How to Build the Demo Application

Now you can build your application.

Steps 1 and 2 are optional. Follow these steps if you want to specify additional build options such as to stop the build process on errors and to keep temporary files that are generated during a build.

1. From the `Build` menu, select `Options...`

   The `Build Options` dialog appears.



☑ Use TASKING build and error parser settings
☑ Save file(s) before starting a command
☐ Scan dependencies before starting a build
☑ Stop build process on error
☐ Keep temporary files that are generated during a build
☐ Use external makefile (instead of 'demo.mak') :
☐ Use additional make options:
Output directory (instead of project directory) :

2. Make your changes and press the `OK` button.

3. From the `Build` menu, select `Scan All Dependencies`.

4. Click on the `Execute 'Make' command` button. The following button is the execute Make button which is located in the ribbon bar.

If there are any unsaved files, EDE will ask you in a separate dialog if you want to save them before starting the build.

How to View the Results of a Build

Once the files have been processed you can inspect the generated messages in the `Output` window.

You can see the commands (and corresponding output captured) which have been executed by the build process in the `Build` tab:

```
TASKING program builder vx.yrz   Build nnn SN 00000000
Assembling ”calc.asm”
Linking to ”asm.out”
Creating IEEE-695 absolute file ”asm.abs”
```

How to Start the CrossView Pro Debugger

Once the files have been compiled, assembled, linked, located and formatted they can be executed by CrossView Pro.

To start CrossView Pro:

1. Click on the `Debug application` button. The following button is the Debug application button which is located in the toolbar.

   CrossView Pro is launched. CrossView Pro will automatically download the compiled file for debugging.

How to Load an Application

You must tell CrossView Pro which program you want to debug. To do this:

1. From the `File` menu, select `Load Symbolic Debug Info...`

   The `Load Symbolic Debug Info` dialog box appears.

2. Click `Load`.

How to View and Execute an Application

To view your source while debugging, the Source Window must be open. To open this window:

1. From the `View` menu, select `Source | Source lines`.

   The source window opens.

   Before starting execution you have to reset the target system to its initial state. The program counter, stack pointer and any other registers must be set to their initial value. The easiest way to do this is:

2. From the `Run` menu, select `Reset Target System`.

   To run your application step–by–step:

3. From the `Run` menu, select `Animate`.

The program `calc.abs` is now stepping through the high level language statements. Using the toolbar or the menu bar you can set breakpoints, monitor data, display registers, simulate I/O and much more. See the *CrossView Pro Debugger User's Guide* for more information.

How to Start a New Project

When you first use EDE you need to setup a project space and add a new project:

1. From the `File` menu, select `New Project Space...`

   The `Create a New Project Space` dialog appears.

2. Give your project space a name and then click `OK`.

   The `Project Properties` dialog box appears.

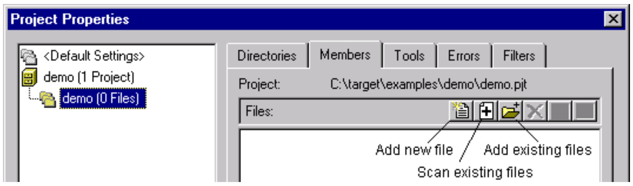
3. Click on the `Add new project to project space` button.

   The `Add New Project to Project Space` dialog appears.

4. Give your project a name and then click `OK`.

   The `Project Properties` dialog box then appears for you to identify the files to be added.

5. Add all the files you want to be part of your project. Then press the `OK` button. To add files, use one of the 3 methods described below.



- If you do not have any source files yet, click on the `Add new file to project` button in the `Project Properties` dialog. Enter a new filename and click `OK`.
- To add existing files to a project by specifying a file pattern click on the `Scan existing files into project` button in the `Project Properties` dialog. Select the directory that contains the files you want to add to your project. Enter one or more file patterns separated by semicolons. The button next to the `Pattern` field contains some predefined patterns. Next click `OK`.

- To add existing files to a project by selecting individual files click on the `Add existing files to project` button in the `Project Properties` dialog. Select the directory that contains the files you want to add to your project. Add the applicable files by double–clicking on them or by selecting them and pressing the `Open` button.

The new project is now open.

6. From the `Project` menu, select `Load Files...` to open the files you want on your EDE desktop.

   EDE automatically creates a makefile for the project. EDE updates the makefile every time you modify your project.

1.4.2 USING THE CONTROL PROGRAM

1. Instead of invoking all the individual translation phases by hand, it is possible (and recommended) to use the control program **cc563** (use **cc56** for the DSP5600x), which calls all phases automatically:

   ```
   cc563 –M –nolib startup.asm calc.asm –o calc.abs
   ```

   As you can see, you may enter multiple input files on the command line. Also, you may specify options and controls for the assembler, linker and locator together. The control program recognizes the options and controls and places them in the appropriate command when invoking the assembler, linker or locator. The control program is described in detail in Chapter 12, *Utilities*.

   The **–M** option specifies to generate map files.

   The **–nolib** option specifies not to link with the standard libraries.

   The **–o** option specifies the name of the output file.

2. If you want to see how the control program calls the assembler, linker and locator, you can use the **–v0** option or **–v** option. The **–v0** option only displays the invocations without executing them. The **–v** option also executes them.

   ```
   cc563 –M –nolib startup.asm calc.asm –o calc.abs –v0
   ```

The control program shows the following command invocations without executing them (UNIX output):

```
startup.asm:
+ as563 –o startup.obj –M24x startup.asm
calc.asm:
+ as563 –o calc.obj –M24x calc.asm
+ lk563 –o/tmp/cc7301b.out –ddef_targ.dsc startup.obj calc.obj
+ lc563 –ocalc.abs –ddef_targ.dsc –f1 –M /tmp/cc7301b.out
```

The **–M24x** option of the assembler selects the 24–bit memory model with default data space in X memory.

3. In step 2, the tools use temporary files for intermediate results. If you want to keep the intermediate files you can use the **–tmp** option. The following command makes this clear.

   ```
   cc563 –M –nolib startup.asm calc.asm –o calc.abs –v0 –tmp
   ```

   This command produces the following output:

   ```
   startup.asm:
   + as563 –o startup.obj –M24x startup.asm
   calc.asm:
   + as563 –o calc.obj –M24x calc.asm
   + lk563 –ocalc.out –ddef_targ.dsc startup.obj calc.obj
   + lc563 –ocalc.abs –ddef_targ.dsc –f1 –M calc.out
   ```

   Assuming the program assembles successfully, the assembler produces the relocatable object modules, `startup.obj` and `calc.obj`.

   Linking and locating the program to absolute addresses is done by two programs: the linker combines objects into a relocatable file with the extension `.out`. The locator binds the program to absolute addresses. The linker takes `a.out` as the default name of the output file. If this name is not suitable, you can specify another filename with the **–o** option.

   Due to the **–M** option, the locator produces the locator map file `calc.map`.

   Besides the output file produced by the linker, the locator can take a so–called description file as input. This file contains a description of the virtual and physical addresses of the program. The chapter *Locator* discusses the exact contents and layout of a description file.

   The result of locator command is the absolute output file `calc.abs`. The file `calc.abs` can be loaded into the CrossView Pro debugger.

1.4.3 USING THE MAKEFILE

The subdirectories in the `examples` directory each contain a `makefile` which can be processed by **mk563** (use **mk56** for the DSP5600x). Also each subdirectory contains a `readme.txt` file with a description of how to build the example.

To build the `calc` demo example follow the steps below. This procedure is outlined as a guide for you to build your own executables for debugging.

1. Make the subdirectory `asm` of the `examples` directory the current working directory.

   This directory contains a makefile for building the `calc` demo example. It uses the default **mk563** rules.

2. Be sure that the directory of the binaries is present in the PATH environment variable.

3. Compile, assemble, link and locate the modules using one call to the program builder **mk563**:

   ```
   mk563
   ```

   This command will build the example using the file `makefile`.

   To see which commands are invoked by **mk563** without actually executing them, type:

   ```
   mk563 -n -a
   ```

   The option **–a** causes all files to be rebuilt, regardless wether they are out of date or not.

   This command produces the following output:

   ```
   TASKING DSP563xx/6xx program builder   vx.yrz Build nnn
   Copyright 1996-year Altium BV          Serial# 00000000
   cc563  -g -M calc.asm -o calc.abs
   ```

   The **–M** option in the makefile is used to create the linker list file (`.lnl`) and the locator map file (`.map`).

   To remove all generated files type:

   ```
   mk563 clean
   ```

1.5 ENVIRONMENT VARIABLES

This section contains an overview of the environment variables used by the DSP56xxx family toolchain.

Environment Variable	Description
AS56INC	Specifies an alternative path for include files for the assembler **as56**.
AS563INC	Specifies an alternative path for include files for the assembler **as563**.
C56INC	Specifies an alternative path for #include files for the C compiler **c56**.
C563INC	Specifies an alternative path for #include files for the C compiler **c563**.
C56LIB	Specifies a path to search for library files used by the linker **lk56**. See also the section *Library Search Path* in the chapter *Linker*.
C563LIB	Specifies a path to search for library files used by the linker **lk563**. See also the section *Library Search Path* in the chapter *Linker*.
CC56BIN	When this variable is set, the control program, **cc56**, prepends the directory specified by this variable to the names of the tools invoked.
CC563BIN	When this variable is set, the control program, **cc563**, prepends the directory specified by this variable to the names of the tools invoked.
CC56OPT	Specifies extra options and/or arguments to each invocation of **cc56**. The control program processes the arguments from this variable before the command line arguments.
CC563OPT	Specifies extra options and/or arguments to each invocation of **cc563**. The control program processes the arguments from this variable before the command line arguments.
LM_LICENSE_FILE	Identifies the location of the license data file. Only needed for hosts that need the FLEXlm license manager.

Environment Variable	Description
PATH	Specifies the search path for your executables.
TMPDIR	Specifies an alternative directory where programs can create temporary files. Used by **c56**, **cc56**, **as56**, **lk56**, **lc56**, **ar56** for the DSP5600x and the **563** versions for the DSP563xx/DSP566xx. See also the next section.

Table 1–1: Environment variables

1.6 TEMPORARY FILES

The assembler, linker, locator and archiver may create temporary files. By default these files will be created in the current directory. If you want the tools to create temporary files in another directory you can enforce this by setting the environment variable TMPDIR.

PC:

```
set TMPDIR=c:\tmp
```

UNIX:

Bourne shell, Korn shell:

```
TMPDIR=/tmp ; export TMPDIR
```

csh:

```
setenv TMPDIR /tmp
```

Note that if you create your temporary files on a directory which is accessible via the network for other users as well, conflicts in the names chosen for temporary files may arise. It is more safe to create temporary files in a directory that is solely accessible to yourself. Of course this does not apply if you run the tools with several users on a multi–user system, such as UNIX. Conflicts may arise if two different computer systems use the same network directory for tools to create their temporary files. For speed reasons a local directory is recommended.

1.7 DEBUGGING WITH CROSSVIEW PRO

To facilitate debugging, you can include symbolic debug information in the load file. During compilation of a high–level–language program, symbolic debug information must be retained that serves as input for the symbolic debugger (**–g** option). The compiler passes symbolic debug information to the assembler by generating SYMB assembler directives in the assembly source file. The assembler translates the SYMB directives to be included in the symbolic debug part of an IEEE–695 object file.

The CrossView Pro debugger (XVW) accepts files with the IEEE–695 format. This is the default output format of the locator. So, you can directly load the file generated by the locator into the CrossView Pro debugger.

The simplest way to build this assembly program ready for debugging is:

```
cc563 –M –nolib startup.asm calc.asm –o calc.abs
```

The result of this command is (output of **–v0** option):

```
startup.asm:
+ as563 –o startup.obj –M24x startup.asm
calc.asm:
+ as563 –o calc.obj –M24x calc.asm
+ lk563 –o/tmp/cc22061b.out –ddef_targ.dsc startup.obj calc.obj
+ lc563 –ocalc.abs –ddef_targ.dsc –f1 –M /tmp/cc22061b.out
```

The control program is described in detail in Chapter 12, *Utilities*.

You can start the debugger for debugging the absolute file `calc.abs` with:

```
xfw56x calc.abs
```

For more information on the debugger, see the *DSP56xxx CrossView Pro Debugger User's Guide*.

The debugger examples are installed in the subdirectory `xvw` of the `examples` directory.

1.8 FILE EXTENSIONS

The extension `.src` or `.asm` is used as input file for the assembler. Files with the extension `.src` are output files of a C compiler. Actually, the assembler accepts files with any extension (or even no extension), but by adding the extension `.asm` to assembler source files, you can distinguish them easily.

If you do not provide a filename extension the assembler will try:

1. the filename itself
2. the filename with `.asm` extension
3. the filename with `.src` extension

So,

```
as563 text
```

only has the same effect as

```
as563 text.asm
```

if the file `text` is not present. In this case, both these commands assemble the file `text.asm` and create a relocatable object module `text.obj`.

For compatibility with future TASKING Cross–Software the following extensions are suggested:

.asm	input assembly source file for **as563**
.src	output from the C compiler **c563** / input for **as563**
.obj	relocatable object files
.a	object library files, output from **ar563**
.out	relocatable output files from **lk563**
.dsc	description file, input for **lc563** and CrossView Pro debugger
.abs	absolute output files from **lc563**

1.9 PREPROCESSING

The assemblers **as56** and **as563** have a built–in macro preprocessor. For a description of the possibilities offered by the macro preprocessor see the chapter *Macro Operations*.

1.10 ASSEMBLER LISTING

The assemblers do not generate a listing file by default. You can generate a listing file with the **–l** option. (See also the **–L** option, for the listing options). As a result of the command:

```
as563 –l text.src
```

the listing file `text.lst` is created.

1.11 ERRORS AND WARNINGS

Any errors detected by the assembler are displayed in the listing file after the actual line containing the error is printed. If no listing file is produced, error messages are still displayed to indicate that the assembly process did not proceed normally.

1.12 COMMAND LINE PROCESSING

This section contains a description of the use of batch files and UNIX scripts. The use of Makefiles is explained in the chapter *Utilities*.

1.12.1 BATCH FILES

Batch files are a facility on the PC whereby one or more commands can be executed from within a file.

Assume that the following sequence of calls is frequently used:

```
cc563 –c ifile.asm –o outfile.obj
```

The files *ifile* and *outfile* may vary from one call to the next. To reduce the number of calls you can make a batch file, for example, `proj.bat`.

Whatever the batch file is called it must end with the file extension `.bat`. The file should contain:

```
cc563 –c %1.asm –o %2.obj
```

On invocation `%1` and `%2` will be replaced by the first and second parameters after the batch file name. Using the name mentioned above (`proj` – note that the file extension `.bat` is not needed for invocation) the call becomes:

```
proj ifile outfile
```

The Windows command prompt will return on the screen the actual command line executed, with all the parameters expanded to the values used.

1.12.2 UNIX SCRIPTS

Scripts are a facility within UNIX whereby one or more commands can be executed from within a file.

Assume that the following sequence of calls is frequently used:

```
cc563 –c ifile.asm –o outfile.obj
```

The files *ifile* and *outfile* may vary from one call to the next. To reduce the number of calls you can make a script, for example, `proj`. The file should contain:

```
cc563 –c $1.asm –o $2.obj
```

On invocation `$1` and `$2` will be replaced by the first and second parameters after the script file name. Using the name mentioned above (`proj`) and after you have set the execute bits of `proj` (`chmod +x proj`) the call becomes:

```
proj ifile outfile
```

CHAPTER 2

ASSEMBLER

TASKING

CHAPTER 2

2.1 DESCRIPTION

The DSP5600x assembler **as56** and the DSP563xx/DSP566xx assembler **as563** are optimizing assemblers. In this chapter these assemblers are all referred to by **as563** unless explicitly stated otherwise. During assembly the assembler builds an internal representation of the program. This representation, the *flow graph*, is used to optimize the program. Examples of the optimizations are parallelization of moves, exchanging instructions and removal of delay slots. After optimization the object file and, optionally, the list file are generated.

The following phases can be identified during assembly:

1. Preprocess, check the syntax and create the flow graph
2. Resolve references to labels in nested scopes (see note)
3. Type determination of all expressions
4. Legality check of all instructions
5. Optimization
6. Address calculation, jump optimization
7. Generation of object and (when requested) list file

As the section information cannot be represented in the IEEE–695 object format the assembler must resolve all label references to labels not contained in the referring section.

The assembler generates relocatable object files using the IEEE–695 object format. This file format specifies a code part and a symbol part as well as a symbolic debug information part.

File inclusion and macro facilities are integrated into the assembler. See the chapter *Macro Operations* for more information.

2.2 INVOCATION

The compiler control program **cc563** may call the assembler automatically. **cc563** translates some of its command line options to options of **as563**. However, the assembler can be invoked as an individual program also.

The **PC** invocation of **as563** is:

> **as563** [*option*]... *source–file* [*map–file*]
> **as563 –V**
> **as563 –?**

When you use a UNIX shell (**C–shell, Bourne shell**), options containing special characters (such as '**( )**') must be enclosed with **" "**. The invocations for UNIX and PC are the same, except for the **–?** option in the **C–shell**:

> **as563 "–?"** or **as563 -\?**

Invocation with **–V** only displays a version header. **–?** shows the invocation syntax.

The *source–file* must be an assembly source file. This file is the input source of the assembler. This file contains assembly code which is either user written or generated by **c563**. Any name is allowed for this file. If this name does not have an extension, the extension `.asm` is assumed or, if the file is still not found, the extension `.src` is assumed.

In the default situation, an object file with extension `.obj` is produced. With the **–l** option a list file with extension `.lst` is produced also. If additionally a *map–file* is specified, only an absolute list file is produced.

Options are preceded by a '–' (minus sign). Options can not be combined after a single '–'. If all goes well, the assembler generates a relocatable object module which contains the object code, with the default extension `.obj`. You can specify another output filename with the **–o** option. Error messages are written to the terminal, unless they are directed to an error list file with the **–err** assembler option.

The following list describes the assembler options briefly. The next section gives a more detailed description.

Option	Description
–?	Display invocation syntax
–D*macro*[**=***def*]	Define preprocessor *macro*
–I*directory*	Look in *directory* for include files
–J[**a**|**l**|**r**]	Select branch mode (**as563** only)
–L[*flag...*]	Select listing file layout
–M*model*	Select memory model: 16–bit, 16/24–bit or 24–bit or DSP566xx (**as563** only)
–O[*flag...*]	Optimization on/off switches
–R[*flag...*]	Remove restrictions
–S[**x**]	Generate Motorola compatible assembly file
–V	Display version header only
–c	Switch to case insensitive mode (default case sensitive)
–e	Remove object file on assembly errors
–err	Redirect error messages to error file
–f *file*	Read options from *file*
–g[*flag...*]	Generate assembly level debug information
–l	Generate listing file
–m*mask*	Select processor mask (**as563** only)
–o *filename*	Specify name of output file
–t	Display section summary
–v	Verbose mode. Print the filenames and numbers of the passes while they progress
–w[*num*]	Suppress one or all warning messages

Table 2–1: Options summary

2.3 DETAILED DESCRIPTION OF ASSEMBLER OPTIONS

With options that can be set from within EDE, you will find a mouse icon that describes the corresponding action.

–?

Option:

–?

Description:

Display an explanation of options at stdout.

Example:

```
as563 –?
```

–c

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Miscellaneous`.
Disable the `Assembler works case sensitive` check box.

–c

Default:

Case sensitive

Description:

Switch to case insensitive mode. By default, the assembler operates in case sensitive mode.

Example:

To switch to case insensitive mode, enter:

```
as563 –c test.src
```

-D

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Miscellaneous`. Define a macro (syntax: *macro*[**=***def*]) in the `Define user macros` field. You can define more macros by separating them with commas.

–D*macro*[**=***def*]

Arguments:

The macro you want to define and optionally its definition.

Description:

Define *macro* as in ’define’. If *def* is not given (’**=**’ is absent), ’1’ is assumed. Any number of symbols can be defined.

Example:

```
as563 –DPI=3.1416 test.src
```

–e

Option:

EDE always removes the object file on errors.

–e

Description:

Use this option if you do not want an object file when the assembler generates errors. With this option the 'make' utility always does the proper productions.

Example:

```
as563 –e test.src
```

-err

Option:

In EDE this option is not useful.

–err

Description:

The assembler redirects error messages to a file with the same basename as the output file and the extension `.ers`. The assembler uses the basename of the output file instead of the input file.

Example:

To write errors to the `test.ers` instead of `stderr`, enter:

```
as563 –err test.src
```

–f

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Miscellaneous`.
Add the option to the `Additional options` field.

–f *file*

Arguments:

A filename for command line processing. The filename ”–” may be used to denote standard input.

Description:

Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.

More than one **–f** option is allowed.

Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

Example:

```
”This has a single quote ’ embedded”
```

or

```
’This has a double quote ” embedded’
```

or

```
’This has a double quote ” and \
a single quote ’”’ embedded”
```

4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non–quoted arguments, all whitespace on the next line will be stripped.

   Example:

```
”This is a continuation \
line”
     –> ”This is a continuation line”

control(file1(mode,type),\
     file2(type))
     –>
control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

Example:

Suppose the file `mycmds` contains the following line:

```
–err
test.src
```

The command line can now be:

```
as563 –f mycmds
```

-g

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Output`.
Enable one or more of the following check boxes:

1. Assembler source line information (excludes 2 and 4)
2. Pass HLL debug information (excludes 1, 3 and 4)
3. Local assembly symbols debug information (excludes 2 and 4)
4. Smart debug (excludes 1, 2 and 3)

–g[*flag*...]

Default:

–gAhLS (only HLL debug)

Description:

Specify to generate debug information. If you do not use this option or if you specify **–g** without a flag, the default is **–gAhLS**, which only passes the high level language debug information.

Flags can be switched on with the lower case letter and switched off with the uppercase letter.

An overview of the flags is given below.

a	– assembler source line information
h	– pass HLL debug information
l	– local symbols debug information
s	– always debug; either ”**AhL**” or ”**aHl**”

With **–ga** you enable assembler source line information. With **–gh** the assembler passes the high level language debug information from the compiler to the object file. These two types of debug information cannot be used both. So, **–gah** is not allowed.

With **–gl** you enable the generation of local symbols debug information. You can use this option independent of the setting of the **–ga** and **–gh** options.

With **–gs** you instruct the assembler to always generate debug information. If HLL debug information is present in the source file, the assembler passes this information (same as **–gAhL**). If no HLL debug information is present, the assembler generates assembler source line information and local symbols debug information (same as **–gaHl**).

Examples:

To pass high level symbolic debug information to the output files and generate local symbols debug information, enter:

```
as563 –ghl test.src
```

To generate assembler source line information, enter:

```
as563 –ga test.src
```

To always generate debug information, depending on the debug information in the source file, enter:

```
as563 –gs test.src
```

–I

Option:

Select the `Project | Directories...` menu item. Add one or more directory paths to the `Include Files Path` field.

–I*directory*

Arguments:

The name of the directory to search for include file(s).

Description:

Change the algorithm for searching include files whose names do not have an absolute pathname to look in *directory*. Thus, include files whose names are enclosed in ”” are searched for first in the directory of the file containing the include line, then in the current directory, then in directories named in **–I** options in left–to–right order. If the include file is still not found, the assembler searches in a directory specified with the environment variable AS563INC (for DSP563xx/DSP566xx), AS56INC (for DSP5600x), AS56INC and AS563INC can contain more than one directory. Separate multiple directories with ’**;**’ for PC (’**:**’ for UNIX). Finally, the directory `../include` relative to the directory where the assembler binary is located is searched.

For include files whose names are in <>, the directory of the file containing the include line and the current directory are not searched. However, the directories named in **–I** options (and the one in AS563INC (AS56INC for DSP5600x), and the relative path) are still searched.

Example:

```
as563 –I/proj/include test.src
```

-J (as563 only)

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Branch Mode`.
Select a branch mode.

–J[**a** | **l** | **r**]

Default:

–J

Description:

Select the branch mode. **–Ja** selects the absolute branch mode and **–Jr** selects the relative branch mode. **–Jl** selects the location–independent branch mode (branches within the source file are made relative, all others absolute). Specifying **–J** or no **–J** option at all, selects the default branch mode (no changes).

Example:

To select the relative branch mode, enter:

```
as563 –Jr test.src
```

–L

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `List File`. Make sure the `Generate list file (.lst)` check box is enabled. Enable or disable one or more check boxes in the `List file generation options` field.

–L[*flag*...]

Arguments:

Optionally one or more flags specifying which source lines are to be removed from the list file.

Default:

–LcDEilMNpQsWXYZ2

Description:

Specify which source lines are to be removed from the list file. A list file is generated when the **–l** option is specified. If you do not specify the **–L** option the assembler removes source lines containing #line directives or symbolic debug information, empty source lines and puts wrapped source lines on one line. **–L** without any flags, is equivalent to **–Lcdelmnpqswxyz**, which removes all specified source lines form the list file.

Flags can be switched on with the lower case letter and switched off with the uppercase letter. The following flags are allowed:

c	<u>Default.</u> Remove source lines containing assembler controls (the OPT directive).
C	Keep source lines containing assembler controls.
d	Remove source lines containing section directives (the ORG directive).
D	<u>Default.</u> Keep source lines containing section directives.
e	Remove source lines containing one of the symbol definition directives EXTERN, GLOBAL, LOCAL or CALLS.
E	<u>Default.</u> Keep source lines containing symbol definition directives.

i Default. Remove source lines included from other files (INCLUDE).

I Keep source lines included from other files (INCLUDE).

l Default. Remove source lines containing C preprocessor line information (lines with #line).

L Keep source lines containing C preprocessor line information.

m Remove source lines containing macro/dup directives (lines with MACRO or DUP).

M Default. Keep source lines containing macro/dup directives.

n Remove empty source lines (newlines).

N Default. Keep empty source lines.

p Default. Remove source lines with false conditional assembly conditions (IF, ELSE, ENDIF).

P Keep source lines with false conditional assembly conditions (IF, ELSE, ENDIF).

q Remove source lines containing assembler equates (lines with EQU or '=').

Q Default. Keep source lines containing assembler equates.

s Default. Remove source lines containing high level language symbolic debug information (lines with SYMB).

S Keep source lines containing HLL symbolic debug information.

w Remove wrapped part of source lines.

W Default. Keep wrapped source lines.

x Remove source lines containing MACRO/DUP expansions.

X Default. Keep source lines containing MACRO/DUP expansions.

y Hide cycle counts.

Y Default. Show cycle counts.

z Show instruction lines from input file.

Z Default. Show actual coded instructions.

0 Hide opcode columns.

1 Show one column of opcodes.

2 Default. Show two columns of opcodes.

Example:

To remove source lines with assembler controls from the resulting list file and to remove wrapped source lines, enter:

```
as563 –l –Lcw test.src
```

–l

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `List File`.
Enable the `Generate list file (.lst)` check box.

–l

Description:

Generate listing file. The listing file has the same basename as the output file. The extension is `.lst`.

Example:

To generate a list file with the name `test.lst`, enter:

```
as563 –l test.src
```

–L

-M (as563 only)

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Miscellaneous`.
Select an `Assembly Mode`.

–M*model*

Arguments:

A memory model:

Model	Description
16	16–bit assembly model (constant expressions)
1624	16/24–bit assembly model
24	24–bit assembly model (default)
6	DSP566xx assembly model

Description:

The DSP566xx assembly model is used to generate DSP566xx compatible assembly. This automatically selects the 16–bit model for constant expressions. It also conforms to the specific pipeline requirements of this processor. This model is only available on the DSP563xx.

The 16–bit model is used to generate constants in 16–bit format for use in the 16–bit arithmetic mode of the processor. It matches the 'opt sbm' assembler directive. This model is only available on the DSP563xx.

The assembler accepts all memory model options (**–M**) from the C compiler (see C manual). The memory model passed on the command line is ignored except for the DSP566xx memory model. The other model options are used to set the predefined functions @MODEL(), @DEFMEM() and @STKMEM() that can be used to make assembly code work in any memory model.

Example:

Assemble a list of fract constants in 16–bit mode:

```
as563 –M16 test.src
```

```
M:ADDRSS CODE   LINE SOURCELINE
                   2
X:000000           3 org   x:
X:000000 000CCD    4 dc    0.1,0.2,0.3
         00199A
         002666
```

-m (as563 only)

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Miscellaneous`. Select a chip mask in the `Select chip mask for problem fixes` field.

–m*mask*

Arguments:

A number indicating the mask for a DSP563xx/DSP566xx processor:

Number	Mask
0	0F92R and 1F92R
1	3F48S

Description:

When the assembler knows different masks for a given processor then you can select a mask using this option. Currently the assembler only supports masks for the DSP563xx processor.

Because of pipeline problems some instructions may not be used after (external) memory accesses, the assembler inserts NOP instructions when necessary.

Example:

To select masks 0F92R and 1F92R, enter:

```
as563 –m0 test.src
```

-O

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Optimization`. Enable or disable one or more `Optimization` check boxes.

–O[*flag*...]

Arguments:

Optionally one or more optimization flags.

Default:

–OGJMNpRS (no optimization)

Description:

Control optimization. **–O** without any flags is the same as specifying **–OgjmnprS**, which performs all optimizations.

Flags can be switched on with the lower case letter and switched off with the uppercase letter. The following flags are allowed:

g Move symbolic debug locations. This option only has effect when used with the **–Om** option. The assembler moves lines containing symbolic debug information to another location in order to perform a better move parallelization.

G <u>Default.</u> Retain symbolic debug locations.

j Enable branch optimization. The assembler tries to replace branches with shorter or faster functionally equivalent branches.

J <u>Default.</u> Disable branch optimization.

m Enable move parallelization. The assembler tries to change any programmed MOVE instruction into a parallel move, added to a previous or next instruction.

M <u>Default.</u> Disable move parallelization.

n Remove existing NOP instructions.

N Default. Do not perform NOP removal.

p Default. Optimize for speed rather than code size.

P Do not perform speed optimization.

r Replace a single instruction DO loop with a REP instruction.

R Default. Do not perform single instruction DO REP optimization.

s Split parallel move instructions in separate instructions before performing optimizations.

S Default. Do not split parallel move instructions.

Example:

To enable move parallelization, enter:

```
as563 –Om test.src
```

Example of backward move parallelization:

```
AND  X0,A1
MOVE (R5)-N5
```

can be changed into:

```
AND  X0,A1 (R5)-N5
```

Example of forward move parallelization:

```
MOVE (R5)-N5
AND  X0,A1
```

can be changed into:

```
AND  X0,A1 (R5)-N5
```

If symbolic debug information was found between instructions that could otherwise be optimized, you also have to specify **–Og**. For example, when you specify **–Ogm** or (**–O**):

```
as563 –Ogm test.src
```

the following code:

```
MOVE (R5)-N5
SYMB ALAB, Fi, #16
AND  X0,A1
```

can be changed into:

```
SYMB ALAB, Fi, #16
AND  X0,A1 (R5)-N5
```

To remove existing NOP instructions, enter:

`as563 -On test.src`

With this option the following code:

```
AND  X0,A1
NOP
AND  Y0,B1
```

can be changed into:

```
AND  X0,A1
AND  Y0,B1
```

To replace a single instruction DO loop with a REP instruction, enter:

`as563 -Or test.src`

With this option the following code:

```
     DO   #$100,label
     MAC  x0,x0,a     x:(r1)+,x0
label:
```

becomes:

```
     REP  #$100
     MAC  x0,x0,a     x:(r1)+,x0
```

To split parallel move instructions before performing optimizations, enter:

`as563 -Os test.src`

With this option the following code:

```
MAC  x0,x0,a     x:(r1)+,x0
```

becomes:

```
MAC  x0,x0,a
MOVE x:(r1)+,x0
```

-o

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Miscellaneous`.
Add the option to the `Additional options` field.

–o *filename*

Arguments:

An output filename. The filename may not start immediately after the option. There must be a tab or space in between.

Default:

Basename of assembly file with `.obj` suffix.

Description:

Use *filename* as output filename of the assembler, instead of the basename of the assembly file with the `.obj` extension.

Example:

To create the object file `myfile.obj` instead of `test.obj`, enter:

```
as563 test.src –o myfile.obj
```

-R

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Optimization`.
Enable or disable one or more `Restrictions` check boxes.

–R[*flag*...]

Arguments:

Optionally one or more remove restriction flags.

Default:

–RDRS (no restrictions are removed)

Description:

Remove pipeline restrictions. Some instruction sequences may cause pipeline effects, as described in section *7.2.2 Summary of Pipeline–Related Restrictions* in Motorola's *DSP56000 Digital Signal Processor Family Manual* [Motorola, inc]. **as563** can remove these restrictions by inserting NOP instructions. **as563** also issues warning W139: "inserted NOP instruction(s) to remove restriction". If a pipeline effect was found and you did not supply any of the **–R** options, **as563** issues warning W140: "previous instruction sequence has a pipeline effect".

–R without any flags is the same as specifying **–Rdrs**, which removes all pipeline restrictions.

Flags can be switched on with the lower case letter and switched off with the uppercase letter. The following flags are allowed:

d Remove DO/ENDDO restrictions. With this option the assembler can check if certain DO/ENDDO instructions may cause problems. For example, a number of operations may not precede a DO loop target label.

D <u>Default.</u> Retain DO/ENDDO restrictions.

r Remove Rn, Nn, Mn pipeline restrictions.

R <u>Default.</u> Retain Rn, Nn, Mn pipeline restrictions.

s Remove stack restrictions. This option removes restrictions that apply to the usage of SSH, SSL, SP, MR and CCR before a RTI or RTS instruction and removes restrictions of SP and SSH/SSL register manipulation.

S Default. Retain stack restrictions.

Example:

To remove DO/ENDDO restrictions and Rn, Nn, Mn pipeline restrictions, enter:

```
as563 –Rdr test.src
```

Example of removing a DO/ENDDO restriction:

```
     do    #10,L1
     movec ssh,x:(R0)+
L1:
```

is changed into:

```
     do    #10,L1
     movec ssh,x:(R0)+
     nop
     nop
     nop
L1:
```

Example of removing a Rn, Nn, Mn restriction:

```
     move  #$100,R7
     move  x:(R7)+,x0
```

is changed into:

```
     move  #$100,R7
     nop
     move  x:(R7)+,x0
```

Example of removing stack restriction:

```
     move  ssh,r0
     RTS
```

is changed into:

```
move  ssh,r0
nop
RTS
```

Section 7.2.2 *Summary of Pipeline–Related Restrictions* in Motorola's *DSP56000 Digital Signal Processor Family Manual* [Motorola, inc]

-S

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Output`.
Enable the `Generate assembly file instead of object file` check box.

–S[**x** | **X**]

Description:

Generate a Motorola compatible assembly file. This option is intended to be used in combination with optimization options of the TASKING assembler to produce an optimized assembly file, which can be used by the Motorola assembler. **–S** is the same as **–SX**. With **–Sx** the assembler does not expand macro/dup definitions in the assembly file.

Example:

The following command produces an optimized Motorola compatible assembly file, called `test.asm`, with expanded macro/dup definitions.

```
as563 –S –Om test.src
```

–t

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Output`.
Enable the `Display module section size summary` check box.

–t

Description:

Produce totals (section size summary). For each section its memory space, size, total cycle counts and name is listed on `stdout`.

The cycle count consists of two parts. The total accumulated count for the section and the total accumulated count for all repeated (REP/DO) instructions. In the case of nested loops it is possible that the total supersedes the section total.

Example:

```
as56  –t test.src
```

```
Section summary:

 Nr M:Loc   Size Cycle/loop Name
  1 P:      000a    24/0    .ptext
  2 X:      0001            .xovl@main
  3 X:      000d            .xstring
```

–V

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Miscellaneous`.
Add the option to the `Additional options` field.

–V

Description:

With this option you can display the version header of the assembler. This option must be the only argument of **as56**. Other options are ignored. The assembler exits after displaying the version header.

Example:

```
as56 –V

TASKING DSP5600x assembler      vx.yrz Build nnn
Copyright 1995–year Altium BV   Serial# 00000000

as563 –V

TASKING DSP563xx/6xx assembler   vx.yrz Build nnn
Copyright 1996–year Altium BV    Serial# 00000000
```

–v

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Miscellaneous`.
Add the option to the `Additional options` field.

–v

Description:

Verbose mode. With this option specified, the assembler prints the filenames and the assembly passes while they progress. So you can see the current status of the assembler.

Example:

`as563 –v test.src`

```
Parsing ”test.src”
   48 lines (total now 43)
Optimizing
Evaluating absolute ORG addresses
Parsing symbolic debug information
Creating object file ”test.obj”
Closing object file
```

–w

Option:

Select the `Project | Project Options...` menu item. Expand the `Assembler` entry and select `Diagnostics`. Select `Display all warnings`, `Suppress all warnings` or `Suppress only certain warnings`.
If you select `Suppress only certain warnings`, type the numbers of the warnings you want to suppress in the corresponding field.

–w[*num*]

Arguments:

Optionally the warning number to suppress.

Description:

–w suppress all warning messages. –**w***num* suppresses warning messages with number *num*. More than one **–w***num* option is allowed.

Example:

The following example suppresses warnings 113 and 114:

```
as563 -w113 -w114 file.src
```

2.4 ENVIRONMENT VARIABLES

AS56INC	With this environment variable you can specify directories where the **as56** assembler will search for include files. You can overrule this search path with the **–I** command line option. Multiple pathnames can be separated with semicolons.
AS563INC	Same as AS56INC, but now for **as563**.
TMPDIR	With the TMPDIR environment symbol you can specify the directory where the assembler can generate temporary files. If the assembler terminates normally, the temporary file will be removed automatically. If you do not set TMPDIR, the temporary file will be created in the current working directory.

2.5 OPTIMIZATIONS

2.5.1 INTRODUCTION

The DSP56xxx assemblers perform various optimizations to speed up assembled applications.

This section discusses the assembler optimizations and their possible implications. All optimizations can be switched on and off from the command line and by using appropriate OPT directive arguments. Some optimizations can only be switched on module basis (last OPT is valid), while others can be changed at any point in the source (e.g. flow).

The following optimizations are available, they are discussed hereafter. See also the **–O** option.

Description	Option	OPT	Module
Move symbolic debug information	–Og	OPHLL	Module
Branch optimization	–Oj	OPJMP	Module
Move parallellization	–Om	OPPM	Module
NOP removal	–On	OPNOP	Flow
Optimize for speed	–Op	OPSPEED	Flow
Single instruction DO to REP	–Or	OPREP	Flow
Split parallel instructions	–Os	OPSP	Module
Retain instruction order		ORDER	Flow
Generic Move instructions		–	–
DO loop code duplication		OPPM *and* OPHLL	Module
Software pipelining		–	–

Table 2–2: Optimizations

2.5.2 MOVE SYMBOLIC DEBUG INFORMATION

All parallellizations are done on instruction blocks surrounded by labels, symbolic debug information, retained NOP instructions or instructions that change the control flow.

In the C compiler generated sources extra debugging information is inserted using SYMB directives. To maintain debugging ease it is possible to instruct the assembler not to move instructions past the debugging information. When debugging ease is not important but code size and speed is, it is possible to instruct the assembler to collect all debugging information at the start of an instruction block. In that case the assembler has more possibilities to move and combine instructions. High level source stepping in a debugger can behave strangely after using this option.

2.5.3 MOVE PARALLELIZATION

As mentioned above the assembler will try to combine instructions within an instruction block. This is done by reordering the instructions to find combinations of instructions that can be combined. The assembler does not try to retain any ordering as written in the assembly source file. The result of the instruction reordering and combination can be seen in the listing file.

2.5.4 BRANCH OPTIMIZATION

The assembler tries to replace branches with shorter or faster functionally equivalent branches. This is only done with branches whose targets are not annotated with the short ('<') or long ('>') operators. Annotated branches remain as specified in the source.

When the **–Oj** option is given the assembler makes extra passes over the flow graph to check whether some branches can be made shorter or faster.

2.5.5 NOP REMOVAL

All NOP instructions (except those immediately following REP instructions) are removed from the program. The assembler tries to reorder the instructions so a minimum of NOP instructions is needed to accommodate for pipeline delays. Sometimes it is necessary to retain NOP instructions. In that case it is possible to enclose those instructions between two OPT directives:

```
OPT   NOOPNOP
MOVE  #$13,X:$FFFC
NOP               ; wait four cycles
NOP               ; these NOPs will not be removed
MOVE  X:$FFFD,X0
OPT   OPNOP
```

2.5.6 OPTIMIZE FOR SPEED

The assembler tries to reduce stall cycles and tries to unroll small loops, at the expense of code size.

2.5.7 SINGLE INSTRUCTION DO LOOPS TO REP

When a DO loop with a body of only one instruction is found then the DO instruction is changed into a REP instruction. In this case the REP instruction is faster and shorter. The drawback is that interrupts are not serviced during a REP loop.

2.5.8 SPLIT PARALLEL INSTRUCTIONS

It is possible to split instructions in single moves and arithmetic instructions before the optimizer tries to reorder and combine the instructions. In some cases this yields shorter code because the optimizer has more combination possibilities.

2.5.9 RETAIN INSTRUCTION ORDER

Sometimes you do not want the assembler to reorder instructions, especially when doing input or output. In this case you can surround the instructions that must be executed in a predetermined order with the OPT ORDER and OPT NOORDER directives. In that case the instructions will retain their ordering, however it is still possible that other instructions can be inserted between or combined with instructions of the sequence. When this is not what you want, you can place a label at the start of and after the instruction sequence, as the assembler does not move instructions past labels.

2.5.10 GENERIC MOVES

Apart from the affected condition flags some MOVE type instructions and arithmetic instructions have almost the same effect. To give the assembler more combination possibilities generic move, GMOVE, instructions may be used. Every move can be written as a GMOVE, but only some of them map into, almost, equivalent TFR or CLR instructions.

Possible mappings are:

GMOVE #$0,*reg*	CLR *reg* MOVE #$0,*reg* (DSP5600x/3xx) MOVEC #$0,*reg*
GMOVE *reg*,*reg*	TFR *reg*,*reg* MOVE *reg*,*reg* MOVEC *reg*,*reg*

Table 2–3: Generic moves

Other GMOVE instructions are replaced by their corresponding MOVE, MOVEC, MOVEP, MOVES or MOVEI counterparts.

2.5.11 DO LOOP CODE DUPLICATION

Sometimes it is possible to combine an instruction in the head of a DO loop body with an instruction in the tail of the DO loop body. When this is possible and when at the target label or directly after the target label a VOID directive is used that lists all registers changed by the instruction, then the instruction is combined with the instruction in the tail of the DO loop and duplicated in front of the DO instruction.

For example:

```
        DO     R0,L1
        MOVE                X:(R2)+,X0
        MAC    X0,X0,A
   L1:  VOID   R2,X0
```

Is changed into:

```
        MOVE                X:(R2)+,X0  ;duplicate move
        DO     R0,L1
        MAC    X0,X0,A      X:(R2)+,X0  ;combined instr.
   L1:  VOID   R2,X0
```

Using DO loop to REP optimization, this can be changed into:

```
        MOVE                X:(R2)+,X0
        REP    R0
        MAC    X0,X0,A      X:(R2)+,X0
```

2.5.12 SOFTWARE PIPELINING

In tight loops the execution speed can often be improved by reshuffling the loop contents and adding code before and after the loop. This allows more code parallelism and decreases pipeline delays. For example, to create an array containing the squares of values in the input array you can code:

```
Farray square:
    do      #20,L1
    move    x:(r0)+,x0
    mpyr    x0,x0,b
    move    b,y(r4)+
L1: void    r0,r4,x0,b
```

Is changed into:

```
Farray square:
    move               x:(r4)+,x0
    mpyr    x0,x0,b    x:(r4)+,x0
    do      #19,L1
    mpyr    x0,x0,b    x:(r4)+,x0     b,y(r0)+
L1:
    move                              b,y(r0)+
```

Using DO loop to REP optimization, this can be changed into:

```
Farray square:
    move               x:(r4)+,x0
    mpyr    x0,x0,b    x:(r4)+,x0
    rep     #19
    mpyr    x0,x0,b    x:(r4)+,x0     b,y(r0)+
    move                              b,y(r0)+
```

In fact, one of the loop iterations is taken out of the loop. The assembler then looks for the optimal placement of the loop start and finish and places them back in the resulting code. The loop goes down from three instructions with a pipeline restriction to only one, a gain of a factor four. However, the code size increases slightly despite the improved parallelism.

Restrictions that apply to this optimization are:

– The loop count must be a constant.
– The loop body must contain no more than ten instructions.

- The VOID directive must have been used for registers that can be destroyed due to the optimization.
- In many cases, optimize for speed (OPT OPSPEED) must be selected to allow for a larger code size.

2.6 LIST FILE

The list file is the output file of the assembler which contains information about the generated code. The amount and form of information depends on the use of the **–L** option. The name is the basename of the output file with the extension `.lst`. The list file is only generated when the **–l** option is supplied. When **–l** is supplied, a list file is also generated when assembly errors/warnings occur. In this case the error/warning is given just below the source line containing the error/warning.

From EDE you can enable the list file generation by enabling the `Generate list file` check box in the `Output` tab of the `Project | Assembler Options | Project Options...` menu item.

2.6.1 ABSOLUTE LIST FILE GENERATION

After locating the whole application, an absolute list file can be generated for all assembly source input files with the assembler. To generate an absolute list file from an assembly source file the source code needs to be assembled again with use of the locator map file of the application the assembly source belongs to. See section 11.7, *Locator Output*, how to produce a locator map file.

An absolute list file contains absolute addresses whereas a standard list file contains relocatable addresses.

When a map file is specified as input for the assembler, only the absolute list file is generated when list file generation is enabled with the list file option **–l**. The previously generated object file is not overwritten when absolute list file generation is enabled. Absolute list file generation is only enabled when a map file is specified on the input which contains the filename extension `.map`.

When you want to generate an absolute list file, you have to specify the same options as you did when generating the object file. If the options are not the same you might get an incorrect absolute list file.

Example:

Suppose your first invocation was:

```
as563 –Oj test.src
```

then when you want to generate an absolute list file you have to specify the same option (**–Oj**), the **–l** option and the map file:

```
as563 –Oj –l test.src test.map
```

With this command the absolute list file "test.lst" is created.

2.6.2 PAGE HEADER

The page header consists of four lines.

The first line contains the following information:

– information about assembler name
– version and serial number
– copyright notice

The second line contains a title specified by the TITLE (first page) or STITLE (succeeding pages) directive and a page number.

The third line contains the name of the file (first page) or is empty (succeeding pages).

The fourth line contains the header of the source listing as described in the next section.

Example:

```
DSP563xx/6xx assembler va.b rc SNzzzzzzzz–zzz (C)year TASKING, Inc.
Title for demo use only                                  page   1
/tmp/hello.asm
M:ADDR CODE           CYCLES LINE SOURCELINE
```

2.6.3 SOURCE LISTING

The following line appears in the page header:

```
M:ADDR CODE            CYCLES LINE SOURCELINE
```

The different columns are discussed below.

M:ADDR This is the memory space and location counter. The memory space can be one of **P**, **X**, **Y** or **L**. The location counter is a (4 digit for DSP5600x, 6 digit for DSP563xx/DSP566xx) hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section.

In lines that generate object code, the value is at the beginning of the line. For any other line there is no display.

Example:

```
M:ADDR CODE           CYCLES LINE SOURCELINE
                                .
                                .
X:0000                         44         org     x,".xovl@malloc",overlay:
                               45         local   ss0000
X:0000                         46 ss0000: ds      49
   |   RESERVED
X:0030
P:0000                         48         org     p,".ptext":
                               49         global  Fmalloc
                               54 Fmalloc:
P:0000 05703C rrrrrr  4    4   55         movec   ssh,x:ss0000
P:0002 200003         2    6   56         tst     a
P:0003 0AF0A2 rrrrrr  6   12   57         jne     L3
```

CODE This is the object code generated by the assembler for this source line, displayed in hexadecimal format. By default two columns are present, but you change this with the **–L1** or **–L0** assembler option to display one code column or even no code column.

The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code or a relocatable part and external part. In this case 'rrrrrr' is printed instead of the value.

For lines that allocate space (DS) the code field contains the text "RESERVED".

Example (as563):

```
M:ADDR   CODE           CYCLES LINE SOURCELINE
                                 .
                                 .
P:000000                         4         org   p:
    |    RESERVED
P:00007E
                                 5
P:00007F 000000         1    1   6         nop
                                 7         align cache
```

In this example the word ”RESERVED” marks the gap generated by the **align cache** directive.

CYCLES The first number is the number of instruction cycles needed to execute the instruction as generated in the CODE field. The second number is the accumulated cycle count of this section. A number within parentheses is the number of cycles of the previous DO loop.

Example:

```
M:LOC  CODE           CYCLES LINE SOURCELINE
                           .
                           .
P:0000 60F400 rrrrrr  4    4   28 Fmain:  move    #L3,r0
P:0002 0AF080 rrrrrr  6   10   29         jmp     Fputs
```

Example (as563):

```
M:ADDR   CODE           CYCLES LINE SOURCELINE
P:000000 60F400 000300  2    2    3        move  #$300,r0
                        3    5    4 ;            (stall 3 cycles)
P:000002 56D800         1    6    4        move  x:(r0)+,a
```

Due to instruction pipelining the instruction on line 4 will stall 3 cycles prior to execution. This kind of stalling can be seen on many places, please refer to the Motorola documentation on the pipeline behavior of the DSP563xx processor. The assembler only shows the pipeline stalling information when the cycle count is also shown (use **opt cc** or the **–LY** command line option).

LINE This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line. If listing of the line is suppressed (i.e. by NOLIST), the number increases by one anyway.

Example:

The following source part,

```
          ;Line 12
     NOLIST
          ;Line 14
     LIST
          ;Line 16
```

results in the following list file part (assemled with **–LC**):

```
M:ADDR CODE           CYCLES LINE SOURCELINE
                                .
                                .
                               12          ;Line 12
                               15      LIST
                               16          ;Line 16
```

SOURCELINE

This column contains the source text. This is a copy of the source line from the source module. For ease of reading the list file, tabs are expanded with sufficient numbers of blank spaces.

If the source column in the listing is too narrow to show the whole source line, the source line is continued in the next listing line.

Errors and warnings are included in the list file following the line in which they occurred.

Example:

```
M:ADDR CODE           CYCLES LINE SOURCELINE
                                .
                                .
                               17         MOVE X0,ABYTE
as563 E217: /tmp/tst.src line 17 : invalid parallel move
as563 W118: /tmp/tst.src line 17 : inserted ”extern ABYTE”
```

2.6.4 OPTIMIZATIONS IN SOURCE LISTING

When optimizing the source file, the assembler moves and combines instructions. The listing file tries to resemble as much as possible the generated object code. For this the source lines are reordered and changed. The line number associated with a source line in the listing resembles the line number of the line in the source file. As the lines are reordered the line numbers are not strictly increasing. The following examples make things clearer:

Parallelized Instructions

```
M:ADDR CODE           CYCLES LINE SOURCELINE
P:0095 21C52A                688       asr     b
                                 ; (  692)     move    a,x1
P:0096 21AF00                689       move    b1,b
P:0097 21E67C                691       sub     y1,b
                                 ; (  690)     move    b,y0
```

Instructions that are combined with other instructions are shown below the latter instruction. These parallelized instructions are preceded by their line number between parentheses.

Inserted NOP Instructions

```
M:ADDR CODE           CYCLES LINE SOURCELINE
P:011D 60F000 rrrrrr         877       move    x:ss_psearch+862,r0
P:011F 000000                877 ;     nop     (inserted)
P:0120 205800                879       move    (r0)+
```

When optimizing the assembler removes all NOP instructions and re–inserts them when it cannot reorder the source so that the NOP would be obsolete. Assembler generated NOP instructions have the line number of the instruction causing the restriction and are marked with the text ”nop (inserted)”.

Labels Combined with Instructions

```
M:ADDR CODE           CYCLES LINE SOURCELINE
P:0115 576600                871       move    b,x:(r6)
                             872 L79:
P:0116 61F000 rrrrrr         873       move    x:ss_psearch+863,r1
P:0118 56F400 000001         872       move    #>1,a
```

When an instruction that was placed after a label is moved to another place the assembler tries to avoid confusion by removing the label before the instruction and removing the instruction after the actual label position. In the example above the line 872 is split in two different lines. Sometimes the assembler cannot construct the source belonging to the actual object, in such cases a label or instruction is mentioned two or more times in the listing file. It is guaranteed that the first occurrence of a label in the listing file is the actual label position. All other occurrences are ghost–occurrences and are not actually generated in the object file.

CHAPTER

3

SOFTWARE CONCEPT

TASKING

CHAPTER 3

3.1 INTRODUCTION

Complex software projects often are divided into smaller program units. These subprograms may be written by a team of programmers in parallel, or they may be programs written for a previous development effort that are going to be reused. The TASKING assembler provides directives to subdivide a program into smaller parts, modules. Symbols can be defined local to a module, so that symbol names can be used without regard to the symbols in other modules. Code and data can be organized in separate sections. These sections can be named in such a way that different modules can implement different parts of these sections. These section can be located in memory by the locator so that concerns about memory placement are postponed until after the assembly process. By using separate modules, a module can be changed without re–assembling the other modules. This speeds up the turnaround time during the development process.

3.2 MODULES

Modules are the separate implementation parts of a project. Each module is defined in a separate file. A module is assembled separately from other modules. By using the INCLUDE directive common definitions and macros can be included in each module. Using the **mk563** utility the module file and include file dependencies can be specified so only the correct modules are re–assembled after changes to one of the files the modules depend upon.

3.2.1 MODULES AND SYMBOLS

A module can use symbols defined in other modules and in the module itself. Symbols defined in a module can be local (other modules cannot access it) or global (other modules have access to it). Symbols outside of a module can be defined with the EXTERN directive. Local symbols are symbols defined by the LOCAL directive, '_'–local labels (underscore labels) or symbols defined with an SET, GSET or EQU directive. Global symbols are either normal labels (non '_'–labels), or symbols explicitly defined global with the GLOBAL directive.

The '_'–labels have their own scoping rules. The scope of such a label is bounded by the surrounding non '_'–label definitions. The TASKING assembler also supports the scoping as imposed by the SECTION and ENDSEC directives, see below for a description. The linker checks the definition of symbols with their reference. For example, when a symbol is defined in a module as a label in P memory and is referred to from another module as a label in X memory an error message will be given. The assembler will perform these checks per module.

3.3 SECTIONS

Sections are relocatable blocks of code and data. Sections are defined with the ORG directive and can be named. A section may have attributes to instruct the locator to place it on a predefined starting address, in near or internal memory or that it may be overlaid with another section. See the ORG directive discussion for a complete description of all possible attributes. Different ORGs with the same name designate the same section, so the attributes of all these ORGs must match. The linker will check this between different modules and emits an error message if the attributes do not match. The linker will also concatenate all matching section definitions into one section. So, all ".text" sections generated by the compiler will be linked into one big ".text" chunk which will be located in one piece. By using this naming scheme it is possible to collect all pieces of code or data belonging together into one bigger section during the linking phase. An ORG directive referring to an earlier defined section is called a continuation. Only the memory type, optional name and optional location counter can be mentioned.

3.3.1 SECTION NAMES

The assembler generates object files in relocatable IEEE–695 object format. The assembler groups units of code and data in the object file using sections. All relocatable information is related to the start address of a section. The locator assigns absolute addresses to sections. A section is the smallest unit of code or data that can be moved to a specific address in memory after assembling a source file. The compiler requires that the assembler supports several different sections with appropriate attributes to assign specific characteristics to those sections. (section with read only data, sections with code, etc.)

ORG *mem*[[**,***name*][**,***attrib*]...]**:**[*abs–loc*]

A section must be declared before it can be used. The ORG directive declares a section with its attributes. A section name can be any identifier. The '@' character is not allowed in regular section names. The assembler and linker use this character to create overlayable sections. This is explained below.

The memory spaces can be:

mem : This defines in which memory space (**X**, **Y**, **L**, **P** or **E**) the section is located. Currently the **E** memory space is not supported.

The attributes can be:

attrib	**Description**
FAR	long addressable
NEAR	short addressable
INTERNAL	internal memory, same as mapping 'I'
EXTERNAL	external memory, same as mapping 'E'
OVERLAY	section must have an overlay name, implies 'scratch'
ABSOLUTE	obsolete, the absolute–location must be an absolute expression
BSS	clear section during startup
CONST	initialize during download, do not generate copy table entry
INIT	initialize section during startup (this attribute is required for P data sections)
MAX	common, overlay with other parts with the same name, is implicitly a type of 'scratch' and 'overlay'
SCRATCH	not filled, not cleared on startup. Section can only contain DS directives, no DC or the like

Table 3–1: Section attributes

Unless disabled, the startup code in the tool chain has to clear BSS sections. These sections contain data space allocations for which no initializers have been specified. BSS sections are zeroed (cleared) at program startup. Sections can be excluded from this initialization with the SCRATCH attribute.

The INIT attribute defines that the section contains initialization data, which is copied from ROM to RAM at program startup. Sections with the CONST attribute, however, are initialized during download.

By default P sections are executable and data memory sections are initialized. The INIT, BSS, CONST and SCRATCH attributes are mutually exclusive. The following table shows the effect of these attributes on section initialization.

Attribute	P section	X, Y, L section
–	executable code, download only	data, init on startup from copy table
INIT	data, init on startup from copy table	data, init on startup from copy table
BSS	data, clear on startup	data, clear on startup
CONST	data, download only	data, download only
SCRATCH	data, no action	data, no action

Table 3–2: Attribute effect on sections

Sections with the NEAR attribute must be allocated in the first 64 words of memory of the DSP56xxx. The locator produces a warning if a section with the NEAR attribute cannot be allocated in this area.

The MAX attribute changes the way the linker determines the section size. Normally the linker determines the section size by accumulating the contents and the sizes of sections with the same name in different object modules. When sections with the same name occur in different object modules with the MAX attribute, the linker generates a section of which the size is the maximum of the sizes in the individual object modules. The MAX attribute applies to BSS sections only.

A section becomes overlayable by specifying the OVERLAY attribute. Only BSS sections are overlayable. The assembler reports an error if it finds the attribute combined with sections of other types. Because it is useless to initialize overlaid sections at program startup time (code using overlaid data cannot assume that the data is in the defined state upon first use), the SCRATCH attribute is defined implicitly when OVERLAY is specified. Overlayable section names are composed as follows:

```
ORG     X,”OVLN@nfunc”, BSS, OVERLAY, NEAR:
            ^    ^
            |    |
   pool name   function name
```

The linker overlays sections with the same pool name. To decide whether BSS sections can be overlaid, the linker builds a call graph. Data in sections belonging to functions that call each other cannot be overlaid. The compiler generates pseudo instructions (CALLS) with information for the linker to build this call graph. The CALLS pseudo has the following syntax:

CALLS '*caller_name*', '*callee_name*' [, '*callee_name*']...

If the function `main()` has overlayable data allocations in the zero page and calls `nfunc()`, the following sections and call information will be generated:

```
ORG X,"OVLN@nfunc", BSS, OVERLAY, NEAR:
ORG X,"OVLN@main", BSS, OVERLAY, NEAR:

CALLS 'main', 'nfunc'
```

This type of overlaying is done by the linker using a call graph, but it is also possible to specify overlaying by the locator using the DELFEE language. See the DELFEE keywords `contiguous` and `overlay` in Appendix G for more information.

Sections become absolute when an address has been specified in the declaration. The assembler generates information in the object file which instructs the locator to put the section contents at the specified address. It is not allowed to make an overlayable section absolute. The assembler reports an error if an absolute location (*abs–loc*) is used in combination with the OVERLAY section attribute.

After a section has been declared, it can be re–activated with the ORG directive:

```
     ORG     X,".STRING",FAR:
     ORG     X,".STRING":
_l001:  DCB  "hello world"
```

All instructions and pseudos which generate data or code must be within an active section. The assembler emits a warning if code or data starts without a section definition and activation.

For reasons of compatibility with the Motorola CLAS assembler the TASKING assemblers for the DSP56xxx also accepts the Motorola CLAS section definition. See the description of the ORG directive in the chapter *Assembler Directives* for more information.

3.3.2 ABSOLUTE SECTIONS

Absolute sections (i.e. ORG directives with a start address) may only be continued in the defining module (continuation). When such a section is defined in the same manner in another module, the locator will try to place the two sections at the same address. This results in a locator error. When an absolute section is defined in more than one module, the section must be defined relocatable and its starting address must be defined in the locator description (`.dsc`) file. Overlay sections may not be defined absolute.

3.3.3 SECTION EXAMPLES

Some examples of the ORG directive are as follows:

ORG `P:$1000`

Defines a section in P memory starting on address $1000. The section is nameless. Other parts of the same section, and in the same module, must be defined with:

```
ORG  P:
```

ORG `X,”.xabs”:$40`

Defines a named section in X memory. The section starts on address $40 and has the name ”.xabs”. Other parts of the same section, that must be in the same module, must be defined with:

```
ORG  X,”.xabs”:
```

ORG `P,”.text”:`

Defines a relocatable named section in P memory. Other parts of this section, with the same name, may be defined in the same module or any other module. These parts use the same ORG statement. When necessary, it is possible to give the section an absolute starting address with the locator description file.

ORG `X,”.xdata”,BSS:`

Defines a relocatable named section in X memory. The BSS attribute instructs the locator to clear the memory located to this section. When this section is used in another module it must be defined identically. Continuations of this section in the same module are as follows:

```
     ORG  X,”.xdata”:
```

ORG `X,”.xovl@f”,OVERLAY:`

Defines a relocatable section in X memory. The section may be overlaid with other overlayable X sections. The function associated with this overlayable part is ”f”. This is the name that should be used with the CALLS directive to designate which function call each other so the linker can build a correct call graph. See also the section *Section Names* above.

3.4 SCOPES

The assembler also supports, on module level, the scoping mechanism as introduced by the Motorola SECTION directive. A block enclosed in a SECTION/ENDSEC pair is called a scope. Scopes are called sections by Motorola, but we use the term section to designate a relocatable block of code or data, as defined by an ORG directive. Symbols defined within a scope are only accessible from within the scope they are defined in or from scopes nested within the defining scope. Exceptions are symbols that, within a scope, are defined GLOBAL. They can be referenced from any other scope.

3.4.1 SCOPE EXAMPLE

The scoping rule can be understood by thoroughly examining the following code fragment, in the comment there is a reference to which symbol the instruction is referencing. Symbol references are described by the construction ”scope.symbol”.

```
; @(#)RESOLVE.ASM 1.1  95/06/26
; LABEL RESOLVING TEST
;
; This should assemble without errors or warnings
;
     ORG    P:

     LOCAL  BW         ; (BW local to this module,
                       ;  not exported to object)
BW:                    ; (Label on module level,
                       ;  defined local, will not
                       ;  be exported to object)
     JMP    S_FW       ; S.S_FW    (was defined global within
                       ;            scope S)
     JMP    FW         ; FW        (forward reference to label
                       ;            on module level)
     JMP    TWICE      ; TWICE     (forward reference to label
                       ;            on module level)

     SECTION      S
     GLOBAL       S_FW
S_FW                   ; = S.S_FW  (global label, will be promoted
                       ;            to module level, is exported
                       ;            as S_FW)
     JMP    TWICE      ; S.TWICE   (forward reference to section
                       ;            label)
TWICE                  ; = S.TWICE (definition hides global symbol
                       ;            TWICE)
     JMP    TWICE      ; S.TWICE   (backward reference to section
                       ;            label)
FW                     ; = S.FW    (definition hides global symbol
                       ;            FW)
     JMP    FW         ; S.FW      (backward reference to section
                       ;            label)
     JMP    BW         ; BW        (backward reference to label
                       ;            on module level)
     ENDSEC

TWICE                  ; = TWICE   (label on module level, is
                       ;            exported by this module)
     JMP    TWICE      ; TWICE     (backward reference)
FW                     ; = FW      (label on module level, is
                       ;            exported by this module)
     JMP    BW         ; BW        (backward reference)
```

```
; AND NOW SOME STACKING OF SECTIONS....

     JMP   LAB           ; LAB       (forward reference to label
                         ;            on module level)
     SECTION    P
     JMP   LAB           ; P.LAB     (forward reference to label
                         ;            defined in the current scope)
     SECTION    Q
     JMP   LAB           ; Q.LAB     (forward reference to local
                         ;            defined in the current scope)
     SECTION    R
     JMP   LAB           ; P.LAB     (Q.LAB is local so invisible
                         ;            to scope R, P.LAB is visible)
     ENDSEC
     ENDSEC
LAB                      ; = P.LAB
     ENDSEC


     SECTION    Q
     LOCAL LAB           ;           (Define lab local to scope Q,
                         ;            invisible from other scopes)
LAB                      ; = Q.LAB   (Local scope label LAB)
     ENDSEC
LAB                      ; = LAB     (Global label LAB, exported
                         ;            to object)


; AND NOW SOME '_'-LABELS:

_LAB  JMP   _LAB         ; Link to this _LAB
                         ; (_LAB is not exported to object)
NO_LAB      JMP   _LAB   ; Link to next _LAB
                         ; (NO_LAB is exported to object,
                         ;  previous _LAB is forgotten)
_LAB                     ; (New definition of _LAB)
     END
```

3.4.2 SCOPES AND SYMBOL NAMES

Symbols defined in a scope are prefixed with the scope name, when these symbols are written to the object file they are transformed into legal assembler symbols.

For example:

```
     SECTION  sec
label:                 ; defines sec.label
     ENDSEC
```

The symbol ”sec.label” will be transformed to the symbol ”sec_label” before it is written to the object file. This can give name collisions with an already defined symbol ”sec_label”. In that case, after renaming, the string ”_x” will be repeatedly appended to the symbol name until the name is unique within the current module. An exception to this rule is symbols that are defined GLOBAL within a scope. They are promoted to the module level, outside any other scope, and placed in the object file as–is, without any pre– or post–fixing of section names or ”_x” strings.

CHAPTER

ASSEMBLY LANGUAGE

4

TASKING

CHAPTER 4

4.1 INPUT SPECIFICATION

An assembly program consists of zero or more statements, one statement per line. A statement may optionally be followed by a comment, which is introduced by a semicolon character (;) and terminated by the end of the input line. Any source statement can be extended to one or more lines by including the line continuation character (\) as the last character on the line to be continued. The length of a source statement (first line and any continuation lines) is only limited by the amount of available memory. Upper and lower case letters are considered equivalent for assembler mnemonics and directives, but are considered distinct for labels, symbols, directive arguments, and literal strings.

A *statement* can be defined as:

[*label*[**:**]] [*instruction* | *directive* | *macro_call*] [**;***comment*]

where,

label is an *identifier*. A space or tab as the first character on a line indicates that the line has no label. There is, however, one exception: the occurrence of *label*: (a symbol followed by a colon) defines a label which may be indented. A label starting with an underscore '_' is a **local label**.

identifier can be made up of letters, digits and/or underscore characters (**_**). The first character may not be a digit. The size of an identifier is only limited by the amount of available memory.

Example:

```
LAB1:     ;This is a label
```

instruction is any valid DSP56xxx assembly language instruction consisting of a mnemonic and one, two, three or no operands and optionally one or two data transfer fields (X and Y parallel move fields). Operands are described in the chapter *Operands and Expressions*. The instructions are described separately in the chapter *Instruction Set*.

Examples:

```
RTI                         ; No operand
INC   B                     ; One operand
ADD   X0,A                  ; Two operands
JSSET #$17,Y:<$3F,$100      ; Three operands
NEG B X1,X:(R3)+ Y:(R6)–,A
  ; one operand and two parallel moves
```

directive any one of the assembler directives; described separately in the chapter *Assembler Directives*.

macro_call a call to a previously defined macro. See the chapter *Macro Operations*.

A statement may be empty.

4.2 ASSEMBLER SIGNIFICANT CHARACTERS

There are several one and two character sequences that are significant to the assembler. Some have multiple meanings depending on the context in which they are used. Special characters associated with expression evaluation are described in Chapter 5, *Operands and Expressions*. Other assembler–significant characters are:

;	–	Comment delimiter
\	–	Line continuation character or Macro dummy argument concatenation operator
?	–	Macro value substitution operator
%	–	Macro hex value substitution operator
^	–	Macro local label override operator
”	–	Macro string delimiter or Quoted string **DEFINE** expansion character
@	–	Function delimiter
*	–	Location counter substitution
++	–	String concatenation operator
[]	–	Substring delimiter

<< –	I/O short addressing mode force operator
< –	Short addressing mode force operator
> –	Long addressing mode force operator
# –	Immediate addressing mode operator
#< –	Immediate short addressing mode force operator
#> –	Immediate long addressing mode force operator

Individual descriptions of each of the assembler special characters follow. They include usage guidelines, functional descriptions, and examples.

;

Comment Delimiter Character

Any number or characters preceded by a semicolon (**;**), but not part of a literal string, is considered a comment. Comments are not significant to the assembler, but they can be used to document the source program. Comments will be reproduced in the assembler output listing. Comments are preserved in macro definitions.

Comments can occupy an entire line, or can be placed after the last assembler–significant field in a source statement. The comment is literally reproduced in the listing file.

Examples:

```
; This comment begins in column 1 of the source file

LOOP JSR COMPUTE  ; This is a trailing comment
     ; These two comments are preceded
     ; by a tab in the source file
```

\

Line Continuation Character or Macro Dummy Argument Concatenation Operator

Line Continuation

The backslash character (\), if used as the last character on a line, indicates to the assembler that the source statement is continued on the following line. The continuation line will be concatenated to the previous line of the source statement, and the result will be processed by the assembler as if it were a single line source statement. The maximum source statement length (the first line and any continuation lines) is 512 characters.

Example:

```
; THIS COMMENT \
EXTENDS OVER \
THREE LINES
```

Macro Argument Concatenation

The backslash (\) is also used to cause the concatenation of a macro dummy argument with other adjacent alphanumeric characters. For the macro processor to recognize dummy arguments, they must normally be separated from other alphanumeric characters by a non–symbol character. However, sometimes it is desirable to concatenate the argument characters with other characters. If an argument is to be concatenated in front of or behind some other symbol characters, then it must be followed by or preceded by the backslash, respectively.

Section 6.5.1.

Example:

Suppose the source input file contained the following macro definition:

```
SWAP_REG  MACRO REG1,REG2  ;swap REG2,REG2
    MOVE  R\REG1,X0        ;using X0 as temp
    MOVE  R\REG2,R\REG1
    MOVE  X0,R\REG2
    ENDM
```

The concatenation operator (\) indicates to the macro processor that the substitution characters for the dummy arguments are to be concatenated in both cases with the character R. If this macro were called with the following statement,

```
SWAP_REG  0,1
```

the resulting expansion would be:

```
MOVE  R0,X0
MOVE  R1,R0
MOVE  X0,R1
```

?

Return Value of Symbol Character

The **?***symbol* sequence, when used in macro definitions, will be replaced by an ASCII string representing the value of *symbol*. This operator may be used in association with the backslash (\) operator. The value of *symbol* must be an integer (not floating point).

Section 6.5.2.

Example:

Consider the following macro definition:

```
SWAP_SYM    MACRO REG1,REG2  ;swap REG1,REG2
      MOVE  R\?REG1,X0       ;using X0 as temp
      MOVE  R\?REG2,R\?REG1
      MOVE  X0,R\?REG2
      ENDM
```

If the source file contained the following SET statements and macro call,

```
AREG  SET       0
BREG  SET       1
      SWAP_SYM  AREG,BREG
```

the resulting expansion as it would appear on the source listing would be:

```
      MOVE  R0,X0
      MOVE  R1,R0
      MOVE  X0,R1
```

%

Return Hex Value of Symbol Character

The **%***symbol* sequence, when used in macro definitions, will be replaced by an ASCII string representing the hexadecimal value of *symbol*. This operator may be used in associations with the backslash (\) operator. The value of *symbol* must be an integer (not floating point).

Section 6.5.3.

Example:

Consider the following macro definition:

```
GEN_LAB  MACRO  LAB,VAL,STMT
LAB\%VAL STMT
      ENDM
```

If this macro were called as follows,

```
NUM   SET     10
      GEN_LAB HEX,NUM,'NOP'
```

The resulting expansion as it would appear in the listing file would be:

```
HEXA  NOP
```

^

Macro Local Label Override

The circumflex (^), when used as a unary operator in a macro expansion, will prevent name mangling of any associated local label. Normally, the macro preprocessor will change any local label inside a macro expansion to a normal label local to the current module. This is done by removing the leading underscore and appending a unique string ”_M_Z*xxxx*” where ”*xxxx*” is a unique sequence number. The ^–operator has no effect on non–local labels or outside of a macro expansion. The ^–operator is useful for passing local labels as macro arguments to be used as referents in the macro. Note that the circumflex is also used as the binary exclusive or operator.

Section 6.5.5.

Example:

Consider the following macro definition:

```
LOAD   MACRO ADDR
       MOVE  P:^ADDR,R0
       ENDM
```

If this macro were called as follows,

```
_LOCAL
       LOAD  _LOCAL
```

the assembler would ordinarily issue an error since _LOCAL is not defined within the body of the macro. With the override operator the assembler retains the _LOCAL symbol and uses that value in the MOVE instruction.

”

Macro String Delimiter or Quoted String DEFINE Expansion Character

Macro String

The double quote (”), when used in macro definitions, is transformed by the macro processor into the string delimiter, the single quote (’). The macro processor examines the characters between the double quotes for any macro arguments. This mechanism allows the use of macro arguments as literal strings.

Section 6.5.4.

Example:

Using the following macro definition,

```
CSTR  MACRO   STRING
      DC      ”STRING”
      ENDM
```

and a macro call,

```
      CSTR    ABCD
```

the resulting macro expansion would be:

```
      DC      ’ABCD’
```

Quoted String DEFINE Expansion

A sequence of characters which matches a symbol created with a **DEFINE** directive will not be expanded if the character sequence is contained within a quoted string. Assembler strings generally are enclosed in single quotes ('). If the string is enclosed in double quotes (") then **DEFINE** symbols will be expanded within the string. In all other respects usage of double quotes is equivalent to that of single quotes.

Example:

Consider the source fragment below:

```
    DEFINE  LONG  ’short’
STR_MAC     MACRO STRING
    MSG     ’This is a LONG STRING’
    MSG     ”This is a LONG STRING”
    ENDM
```

If this macro were invoked as follows,

```
    STR_MAC sentence
```

then the resulting expansion would be:

```
    MSG  ’This is a LONG STRING’
    MSG  ’This is a short sentence’
```

@

Function Delimiter

All assembler built–in functions start with the @ symbol. See section 5.4 for a full discussion of these functions.

Example:

```
SVAL  EQU  @SQT(FVAL)  ; Obtain square root
```

*

Location Counter Substitution

When used as an operand in an expression, the asterisk represents the current integer value of the runtime location counter.

Example:

```
      ORG  X:$100
XBASE EQU  *+$20   ; XBASE = $120
```

++

String Concatenation Operator

Any two strings can be concatenated with the string concatenation operator (**++**). The two strings must each be enclosed by single or double quotes, and there must be no intervening blanks between the string concatenation operator and the two strings.

Example:

```
'ABC'++'DEF' = 'ABCDEF'
```

[ ]

Substring Delimiter

[*string*,*offset*,*length***]**

Square brackets a substring operation. The *string* argument is the source string. *offset* is the substring starting position within *string*. *length* is the length of the desired substring. *string* may be any legal string combination, including another substring. An error is issued if either *offset* or *length* exceed the length of *string*.

Example:

```
DEFINE ID  ['DSP56000',3,5]  ; ID = '56000'
```

<<

I/O Short Addressing Mode Force Operator

Many DSP instructions allow an I/O short form of addressing. If the value of an absolute address is known to the assembler on pass one, then the assembler will always pick the shortest form of addressing consistent with the instruction format. If the absolute address is not known to the assembler on pass one (that is, the address is a forward or external reference), then the assembler will pick the long form of addressing by default. If this is not desired, then the I/O short form of addressing can be forced by preceding the absolute address by the I/O short addressing mode force operator (**<<**).

Example:

Since the symbol IOPORT is an external reference in the following sequence of source lines, the assembler would pick the long absolute form of addressing by default:

```
BTST    #4,Y:IOPORT
EXTERN  IOPORT
```

Because the long absolute addressing mode would cause the instruction to be two words long instead of one word for the I/O short absolute addressing mode, it would be desirable to force the I/O short absolute addressing mode as shown below:

```
BTST    #4,Y:<<IOPORT
EXTERN  IOPORT
```

<

Short Addressing Mode Force Operator

Many DSP instructions allow a short form of addressing. If the value of an absolute address is known to the assembler on pass one, or the **FORCE** NEAR directive is active, then the assembler will always pick the shortest form of addressing consistent with the instruction format. If the absolute address is not known to the assembler on pass one (that is, the address is a forward or external reference), then the assembler will pick the long form of addressing by default. If this is not desired, then the short absolute form of addressing can be forced by preceding the absolute address by the short addressing mode force operator (**<**).

FORCE

Example:

Since the symbol DATAST is an external reference in the following sequence of source lines, the assembler would pick the long absolute form of addressing by default:

```
MOVE    Y0,Y:DATAST
EXTERN  DATAST
```

Because the long absolute addressing mode would cause the instruction to be two words long instead of one word for the short absolute addressing mode, it would be desirable to force the short absolute addressing mode as shown below:

```
MOVE    Y0,Y:<DATAST
EXTERN  DATAST
```

>

Long Addressing Mode Force Operator

Many DSP instructions allow a long form of addressing. If the value of an absolute address is known to the assembler on pass one, then the assembler will always pick the shortest form of addressing consistent with the instruction format, unless the **FORCE** FAR directive is active. If this is not desired, then the long absolute form of addressing can be forced by preceding the absolute address by the long addressing mode force operator (**>**).

FORCE

Example:

Since the symbol DATAST is known in the following sequence of source lines, the assembler would pick the short absolute form of addressing:

```
        MOVE  Y0,Y:DATAST
DATAST  EQU   Y:$23
```

If this is not desirable, then the long absolute addressing mode can be forced as shown below:

```
        MOVE  Y0,Y:>DATAST
DATAST  EQU   Y:$23
```

#

Immediate Addressing Mode

The pound sign (**#**) is used to indicate to the assembler to use the immediate addressing mode.

Example:

```
CNST  EQU   $5
      MOVE  #CNST,X0
```

#<

Immediate Short Addressing Mode Force Operator

Many DSP instructions allow a short form of addressing. If the immediate data is known to the assembler on pass one (not a forward or external reference), or the **FORCE** NEAR directive is active, then the assembler will always pick the shortest form of immediate addressing consistent with the instruction. If the immediate data is a forward or external reference, then the assembler will pick the long form of immediate addressing by default. If this is not desired, then the short form of addressing can be forced using the immediate short addressing mode force operator (**#<**).

FORCE

Example:

In the following sequence of source lines, the symbol CNST is not known to the assembler on pass one, and therefore, the assembler would use the long immediate addressing form for the **MOVE** instruction.

```
      MOVE  #CNST,X0
CNST  EQU   $5
```

Because the long immediate addressing mode makes the instruction two words long instead of one word for the immediate short absolute addressing mode, it may be desirable to force the immediate short addressing mode as shown below:

```
      MOVE  #<CNST,X0
CNST  EQU   $5
```

#>

Immediate Long Addressing Mode Force Operator

Many DSP instructions allow a long immediate form of addressing. If the immediate data is known to the assembler on pass one (not a forward or external reference), then the assembler will always pick the shortest form of immediate addressing consistent with the instruction, unless the **FORCE** FAR directive is active. If this is not desired, then the long form of addressing can be forced using the immediate long addressing mode force operator (**#>**).

FORCE

Example:

In the following sequence of source lines, the symbol CNST is known to the assembler on pass one, and therefore, the assembler would use the short immediate addressing form for the **MOVE** instruction.

```
CNST  EQU   $5
      MOVE  #CNST,X0
```

If this is not desirable, then the long immediate form of addressing can be forced as shown below:

```
CNST  EQU   $5
      MOVE  #>CNST,X0
```

4.3 REGISTERS

The following register names, either upper or lower case, cannot be used as symbol names in an assembly language source file:

```
A    A0   A1   A2   AB
B    B0   B1   B2   BA
X    X0   X1
Y    Y0   Y1

R0   R1   R2   R3   R4   R5   R6   R7
N0   N1   N2   N3   N4   N5   N6   N7
M0   M1   M2   M3   M4   M5   M6   M7

SR   MR   CCR
OMR  EOM  COM
VBA
EP   SC   SZ   SP
SSH  SSL
LA   LC
```

The following registers are used by the assembler in structured control statement processing (Chapter 8):

```
A    X0   Y0   Y1
```

CHAPTER

OPERANDS AND EXPRESSIONS

5

TASKING

CHAPTER 5

5.1 OPERANDS

An operand is the part of the instruction that follows the instruction opcode. There can be one, two, three or even no operands in an instruction. An operand of an assembly instruction has one of the following types:

Operands	Description
expr	any valid expression as described in the section *Expressions*.
reg	any valid register as described in the section *Registers*.
symbol	a symbolic name as created by an equate. A symbol can be an expression.
address	a combination of expr, reg and symbol.

If an expression can be completely evaluated at assembly time, it is called an absolute expression; if it is not, it is called a relocatable expression. See the section 5.2, *Expressions*, for more details.

5.1.1 OPERANDS AND ADDRESSING MODES

The DSP56xxx assembly language has several addressing modes. These are listed below with a short description. For details see the DSP56000, DSP56300 and DSP56600 Family Manuals.

Register Direct

The instruction specifies the register which contains the operand.

Syntax:

mnemonic *register*

Address Register Indirect

The instruction specifies the register containing the operand address. Several forms are available.

Syntax:

mnemonic	(R*n*)	
mnemonic	(R*n*)+	
mnemonic	(R*n*)–	
mnemonic	(R*n*)+N*n*	
mnemonic	(R*n*)–N*n*	
mnemonic	(R*n*+N*n*)	
mnemonic	–(R*n*)	
mnemonic	(R*n*+*displ*)	(not for DSP5600x)

Immediate

An immediate operand is a one word number or short number, which is encoded as part of the instruction. Immediate operands are indicated by the # sign before the expression defining the value of the operand.

Syntax:

mnemonic *#number*

Absolute

The instruction contains the operand address. The address can be 16 or 24 bits (absolute address), 6 bits zero extended (absolute short) or 6 bits ones extended (I/O short).

Syntax:

mnemonic *direct_address*

Short Jump

The instruction contains the 12–bit address zero extended to 16 or 24 bits, which allows addresses $000–$FFF to be accessed.

Syntax:

mnemonic *jump_address*

5.2 EXPRESSIONS

An operand of an assembler instruction or directive is either an assembler symbol, a register name or an expression. An expression is a sequence of symbols that denotes an address in a particular memory space or a number.

Expressions that can be evaluated at assembly time are called **absolute expressions**. Expressions where the result is unknown until all sections have been combined and located are called **relocatable expressions**. When any operand of an expression is relocatable the entire expression is relocatable. Relocatable expressions are emitted in the object file and evaluated by the linker or the locator. Relocatable expressions may only contain integral functions; floating point functions and numbers are not supported by the IEEE object format. An error is emitted when during object creation non–IEEE relocatable expressions are found.

An expression has a type which depends on the type of the identifiers in the expression. See section 5.2.4, *Expression Type*, for details.

The assembler evaluates expressions with 64–bit precision in two's complement.

The syntax of an *expression* can be any of the following:

– *number*
– *expression_string*
– *symbol*
– *expression binary_operator expression*
– *unary_operator expression*
– **(** *expression* **)**
– *function*

Spaces are not allowed inside expressions! Example: `3 + 4` is not valid, but `3+4` is.

All types of expressions are explained below and in the following sections.

( ) You can use parentheses to control the evaluation order of the operators. What is between parentheses is evaluated first.

Examples:

```
(3+4)*5    ; Result is 35.
           ; 3 + 4 is evaluated first.
3+(4*5)    ; Result is 23.
           ; 4 * 5 is evaluated first.
           ; parentheses are superfluous here
```

5.2.1 NUMBER

Numeric constants can be used in expressions. If there is no prefix, the assembler assumes the number is decimal.

number	can be one of the following: – ***%****bin_num* – *dec_num* (or **‘***dec_num*) – **$***hex_num* – *float_num*
bin_num	is a binary number formed of ’0’–’1’ prefixed with a ’%’. **Examples:** `%1001; %1011; %01100100;`
dec_num	is a decimal number formed of ’0’–’9’, optionally prefixed with a ’**‘**’. **Examples:** `12; ‘5978;`
hex_num	is a hexadecimal number formed of the characters ’0’–’9’ and ’a’–’f’ or ’A’–’F’ prefixed with a ’$’. **Examples:** `$45; $FFD4; $9abc`
float_num	is a floating point number formed of ’0’–’9’ and indicated either by a preceding, following, or included decimal point or by the presence of ’E’ or ’e’. Floating point numbers are always base 10. **Examples:** `.12; 5E10; 3.6e8`

A number may be written without a leading radix indicator if the input radix is changed using the RADIX directive. For example, a hexadecimal number may be written without the leading dollar sign (**$**) if the input radix is set to 16 (assuming an initial radix of 10). The default radix is 10.

5.2.2 EXPRESSION STRING

An *expression_string* is a *string* with an arbitrary length evaluating to a number. The value of the string is calculated by taking the first 4 characters padded with 0 to the left.

string	is a string of ASCII characters, enclosed in single (′) or double (″) quotes. The starting and closing quote must be the same. To include the enclosing quote in the string, double it. E.g. the string containing both quotes can be denoted as: ″′″″″ or ′′′″′. See the chapter *Macro Operations* for the differences between single and double quoted strings.

Examples:

```
’A’+1      ; a 1–character ASCII string,
           ; result $00000042
”9C”+1     ; a 2–character ASCII string,
           ; result $00003944
```

Two strings can be concatenated with the strings concatenation operator (**++**). The two strings must each be enclosed by single or double quotes, and there must be no intervening blanks between the operator and the two strings:

```
’AB’++’CD’  ; two 2–character ASCII strings
            ; concatenated (same as ’ABCD’),
            ; result $41424344
```

Square brackets (**[ ]**) delimit a substring operation in the form:

[*string***,***offset***,***length***]**

offset is the start position within *string*. *length* is the length of the desired substring. Both values may not exceed the size of *string*.

```
DEFINE ID [’DSP56000’,3,5]   ; ID = ’56000’
```

5.2.3 SYMBOL

A *symbol* is an *identifier*. A *symbol* represents the value of an *identifier* which is already defined, or will be defined in the current source module by means of a label declaration or an equate directive.

Examples:

```
CON1 EQU $3    ; The variable CON1 represents
               ; the value of 3

MOVE CON1+$20,R1  ; Move contents of address
                  ; $23 to register R1
```

When you invoke one of the assemblers, the following predefined symbol exists:

_AS56	contains a string with the name of the assembler (”as56” or ”as563”)

5.2.4 EXPRESSION TYPE

The type of an expression is either a number (floating point or integral) or an address. The result type of an expression depends on the operator and its operands. The tables below summarize all available operators.

Please note:

1. when an illegal combination (denoted as *) is found, a warning is emitted and the expression type will be undefined;
2. a label is of type ’address’; an equate symbol has the type of the equate expression;
3. the type of an untyped symbol can be an address or a number, depending on the context; the result of the operation can be determined using the tables;
4. the binary logical and relational operators (||, &&, ==, !=, <, <=, >, >=) accept any combination of operands, the result is always the integral number 0 or 1;
5. the binary shift and bitwise operators <<, >>, |, & and ^ only accept integral operands.

The following table shows the result type of expressions with unary operators.

Operator	num	addr
~	num	*
!	num	*
–	num	*
+	num	num

Table 5–1: Expression type, unary operators

The following table shows the result type of expressions with binary numerical operators.

Operator	num, num	addr, num	num, addr	addr,addr
–	num	addr	*	num
+	num	addr	addr	*
*	num	*	*	*
/	num	*	*	*
%	num	*	*	*

Table 5–2: Expression type, binary numerical operators

”num <op> float” and ”float <op> num” evaluates to float

any combination of an address with a float is illegal

a string operand will be converted to an integral number

”addr – addr” is only valid when both addresses are in the same address space, or in compatible address spaces (see Table 5–4)

The following table shows the result type of functions. num can be float and integer. A '–' in the column Operands means that the function has no operands.

Function	Operands	Result
@ABS()	num	float
@ACS()	num	float
@ARG()	symbol integer	integer integer
@AS56()	–	string
@ASN()	num	float
@AT2()	num,num	float
@ATN()	num	float
@CEL()	num	float
@CNT()	–	float
@COH()	num	float
@COS()	num	float
@CVF()	num	float
@CVI()	num	integer
@CVS()	mem,expr	mem:expr
@DEF()	symbol	integer
@DEFMEM()	–	string
@DSP()	–	integer
@FLD()	num,num,num num,num,num,num	integer integer
@FLR()	num	float
@FRC()	num	integer
@L10()	num	float
@LEN()	string	integer
@LFR()	num	integer
@LNG()	num,num	integer
@LOG()	num	float
@LST()	–	integer
@LUN()	num	float

Function	Operands	Result
@MAC()	symbol	integer
@MAX()	num,num,...,float,... integer,integer,...	float integer
@MIN()	num,num,...,float,... integer,integer,...	float integer
@MODEL()	–	string
@MSP()	num	integer
@MXP()	–	integer
@POS()	string,string string,string,num	integer integer
@POW()	num,num	float
@RND()	–	float
@RVB()	num num,num	integer integer
@SCP()	string,string	integer
@SGN()	num	integer
@SIN()	num	float
@SNH()	num	float
@SQT()	num	float
@STKMEM()	–	string
@TAN()	num	float
@TNH()	num	float
@UNF()	num	float
@XPN()	num	float

Table 5–3: Expression type, functions

5.2.5 MEMORY SPACES

All addresses have a memory space attribute of either **X**, **Y**, **L**, **P**rogram, **E**MI, or **U**nknown, to distinguish between different address spaces. A number always has the memory attribute **N**one. When subtracting an address from another address the address spaces are combined to form the resulting type. See the table below for legal combinations and their resulting address space. The type **N** stands for a number. The function @CVS() can be used to change the address space of an address or to make an address of any number. The @MSP() function can be used to extract the address space from any expression. An external symbol which is declared without a memory space can be used as an address in any memory space.

		Left Operand Memory Space Attribute					
		N	**X**	**Y**	**L**	**P**	**E**
Right Operand Memory Space Attribute	**N**	N	X	Y	L	P	E
	X	*	N	*	N	*	*
	Y	*	*	N	N	*	*
	L	*	N	N	N	*	*
	P	*	*	*	*	N	*
	E	*	*	*	*	*	N

* = Represents an illegal operation that will result in an error.

Table 5–4: Compatible memory types for binary '–' operator

Memory space attributes become important when an expression is used as an address. Errors will occur when the memory space attribute of the expression result does not match the explicit or implicit memory space specified in the source code. Memory spaces are explicit when the address has any of the following forms:

Attribute	Explicit Address
X	X:*address_expression*
Y	Y:*address_expression*
L	L:*address_expression*
P	P:*address_expression*
E	E:*address_expression*

Table 5–5: Memory space attributes

The memory space is implicitly **P** when an address is used as the operand of a **DO**, branch, or jump–type instruction.

Expressions used for immediate addressing can have any memory space attribute.

5.2.6 EXAMPLE

```
/usr/src/dsp56/op_types.asm
M:LOC  CODE      CYCLES LINE SOURCELINE
P:0200                     1        org     p:$200
                           2 plab:
X:0100                     3        org     x:$100
                           4 xlab:
                           5 xlab2:
X:0100 000100              6        dc      plab–xlab
as56 W123: /usr/src/dsp56/op_types.asm line 6 :
 expression: illegal combination of memory spaces
X:0101 000200              7        dc      xlab+xlab2
as56 W123: /usr/src/dsp56/op_types.asm line 7 :
 expression: arithmetic on an address with an address
                           8
X:0102 000200              9        dc      xlab*2
as56 W123: /usr/src/dsp56/op_types.asm line 9 :
 expression: undefined arithmetic on an address
X:0103 000200             10        dc      @cvs(n,xlab)*2
                                    ; use @CVS() so we can multiply
                          11        end
```

5.3 OPERATORS

There are two types of operators:

– unary operators
– binary operators

Operators can be arithmetic operators, shift operators, relational operators, bitwise operators, or logical operators. All operators are described in the following sections.

If the grouping of the operators is not specified with parentheses, the operator precedence is used to determine evaluation order. Every operator has a precedence level associated with it. The following table lists the operators and their order of precedence (in descending order).

Operators	Type
+, –, ~, !	unary
*, /, %	binary
+, –	binary
<<, >>	binary
<, <=, >, >=	unary
==, !=	binary
&, |, ^	binary
&&, ||	binary

Table 5–6: Operators Precedence List

Except for the unary operators, the assembler evaluates expressions with operators of the same precedence level left–to–right. The unary operators are evaluated right–to–left . So, `-4+3*2` evaluates to `(-4)+(3*2)`.

5.3.1 ADDITION AND SUBTRACTION

Synopsis:

Addition:	*operand*	**+**	*operand*
Subtraction:	*operand*	**–**	*operand*

The + operator adds its two operands and the – operator subtracts them. The operands can be any expression evaluating to an absolute number or a relocatable operand, with the restrictions of Table 5–2.

Examples:

```
$a342+$23        ; addition of absolute numbers
$ff1a–AVAR       ; subtraction with the value of
                 ; symbol AVAR
```

5.3.2 SIGN OPERATORS

Synopsis:

Plus:	**+***operand*
Minus:	**–***operand*

The + operator does not modify its operand. The – operator subtracts its operand from zero. See also the restrictions in Table 5–1.

Example:

```
5+–3  ; result is 2
```

5.3.3 MULTIPLICATION AND DIVISION

Synopsis:

Multiplication:	*operand*	*****	*operand*
Division:	*operand*	/	*operand*
Modulo:	*operand*	**%**	*operand*

The * operator multiplies its two operands, the / operator performs an integer division, discarding any remainder. The % operator also performs an integer division, but discards the quotient and returns the remainder. The operands can be any expression evaluating to an absolute number or a relocatable operand, with the restrictions of Table 5–2. Note that the right operands of the / and % operator may not be zero.

Examples:

```
AVAR*2          ; multiplication
$ff3c/COUNT     ; division
23%4            ; modulo, result is 3
```

5.3.4 SHIFT OPERATORS

Synopsis:

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

These operators shift their left operand (*operand*) either left (<<) or right (>>) by the number of bits (absolute number) specified with the right operand (*count*). The operands can be any expression evaluating to an (integer) number.

Examples:

```
AVAR>>4         ; shift right variable AVAR, 4 times
```

5.3.5 RELATIONAL OPERATORS

Synopsis:

Equal:	*operand*	**==**	*operand*
Not equal:	*operand*	**!=**	*operand*
Less than:	*operand*	**<**	*operand*
Less than or equal:	*operand*	**<=**	*operand*
Greater than:	*operand*	**>**	*operand*
Greater than or equal:	*operand*	**>=**	*operand*

These operators compare their operands and return an absolute number (an integer) of 1 for 'true' and 0 for 'false'. The operands can be any expression evaluating to an absolute number or a relocatable operand.

Examples:

```
3>=4         ; result is 0 (false)
4==COUNT     ; 1 (true), if COUNT is 4.
             ; 0 otherwise.
9<$0A        ; result is 1 (true)
```

5.3.6 BITWISE OPERATORS

Synopsis:

Bitwise AND:	*operand*	**&**	*operand*
Bitwise OR:	*operand*	**|**	*operand*
Bitwise XOR:	*operand*	**^**	*operand*
One's complement		**~**	*operand*

The AND, OR and XOR operators take the bitwise AND, OR respectively XOR of the left and right operand. The one's complement (bitwise NOT) operator performs a bitwise complement on its operand. The operands can be any expression evaluating to an (integer) number.

Examples:

```
$B&3  ; result is 3
          %1011
          %0011  &
          ---------
          %0011
```

```
~$A   ; result is $fffff5
          ~ %00000000 00000000 00001010
          = %11111111 11111111 11110101
```

5.3.7 LOGICAL OPERATORS

Synopsis:

Logical AND:	*operand*	**&&**	*operand*
Logical OR:	*operand*	**||**	*operand*
Logical NOT:		**!**	*operand*

The logical AND operator returns an integer 1 if both operands are non–zero; otherwise it returns an integer 0. The logical OR operator returns an integer 1 if either of its operands is non–zero; otherwise it returns an integer 0. The **!** operator performs a logical not on its operand. **!** returns an integer 1 (‘true) if the operand is 0; otherwise, **!** returns 0 (‘false’). The operands can be can be any expression evaluating to an integer or floating point.

Examples:

```
$B&&3     ; result is 1 (true)

!$A       ; result is 0 (false)
!(4<3)    ; result is 1 (true)
          ; 4 < 3 result is 0 (false)
```

5.4 FUNCTIONS

The assembler has several built–in functions to support data conversion, string comparison, and math computations. Functions can be used as terms in any arbitrary expression. Functions have the following syntax:

*@function_name***(***argument*[**,***argument*]...**)**

Functions start with the '@' sign and have zero or more arguments, and are always followed by opening and closing parentheses. There must be no intervening spaces between the function name and the opening parenthesis and between the (comma–separated) arguments.

Assembler functions can be grouped into five types:

1. Mathematical functions
2. Conversion functions
3. String functions
4. Macro functions
5. Assembler mode functions

5.4.1 MATHEMATICAL FUNCTIONS

The mathematical functions comprise transcendental, random value, and min/max functions, among others:

ABS	–	Absolute value
ACS	–	Arc cosine
ASN	–	Arc sine
AT2	–	Arc tangent
ATN	–	Arc tangent
CEL	–	Ceiling function
COH	–	Hyperbolic cosine
COS	–	Cosine
FLR	–	Floor function
L10	–	Log base 10

LOG	–	Natural logarithm
MAX	–	Maximum value
MIN	–	Minimum value
POW	–	Raise to a power
RND	–	Random value
SGN	–	Return sign
SIN	–	Sine
SNH	–	Hyperbolic sine
SQT	–	Square root
TAN	–	Tangent
TNH	–	Hyperbolic tangent
XPN	–	Exponential function

5.4.2 CONVERSION FUNCTIONS

The conversion functions provide conversion between integer, floating point, and fixed point fractional values:

CVF	–	Convert integer to floating point
CVI	–	Convert floating point to integer
CVS	–	Convert memory space
FLD	–	Shift and mask operation
FRC	–	Convert floating point to fractional
LFR	–	Convert floating point to long fractional
LNG	–	Concatenate to double word
LUN	–	Convert long fractional to floating point
RVB	–	Reverse bits in field
UNF	–	Convert fractional to floating point

5.4.3 STRING FUNCTIONS

String functions compare strings, return the length of a string, and return the position of a substring within a string:

LEN	–	Length of string
POS	–	Position of substring in string
SCP	–	Compare strings

5.4.4 MACRO FUNCTIONS

Macro functions return information about macros:

ARG	–	Macro argument function
CNT	–	Macro argument count
MAC	–	Macro definition function
MXP	–	Macro expansion function

5.4.5 ASSEMBLER MODE FUNCTIONS

Miscellaneous functions having to do with assembler operation:

AS56	–	Assembler executable name
DEF	–	Symbol definition function
DEFMEM	–	Default memory function
DSP	–	DSP processor type number
LST	–	LIST directive flag value
MODEL	–	Returns memory model
MSP	–	Memory space
STKMEM	–	Stack memory function

5.4.6 DETAILED DESCRIPTION

Individual descriptions of each of the assembler functions follow. They include usage guidelines, functional descriptions, and examples.

@ABS(**expression**)

Returns the absolute value of *expression* as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
MOVE  #@ABS(VAL),A   ;load absolute value
```

@ACS(**expression**)

Returns the arc cosine of *expression* as a floating point value in the range zero to pi. The result of *expression* must be between –1 and 1. The memory space attribute of the result will be **N**one.

Example:

```
ACOS = @ACS(–1.0)    ;ACOS = 3.141593
```

@ARG(**symbol | expression**)

Returns integer 1 if the macro argument represented by *symbol* or *expression* is present, 0 otherwise. If the argument is a symbol it must be single–quoted and refer to a dummy argument name. If the argument is an expression it refers to the ordinal position of the argument in the macro dummy argument list. A warning will be issued if this function is used when no macro expansion is active. The memory space attribute of the result will be **N**one.

Example:

```
IF  @ARG(TWIDDLE)    ;twiddle factor provided?
```

@AS56()

Returns the name of the assembler executable. This is **as56** for the DSP5600x or **as563** for the DSP563xx family.

Example:

```
ANAME  DC  @AS56()   ;ANAME = as563 for DSP563xx
```

@ASN(*expression****)***

Returns the arc sine of *expression* as a floating point value in the range –pi/2 to pi/2. The result of *expression* must be between –1 and 1. The memory space attribute of the result will be **N**one.

Example:

```
ARCSINE  SET  @ASN(-1.0)   ;ARCSINE = -1.570796
```

@AT2(*expr1,expr2****)***

Returns the arc tangent of *expr1/expr2* as a floating point value in the range –pi to pi. Expr1 and expr2 must be separated by a comma. The memory space attribute of the result will be **N**one.

Example:

```
ATAN  EQU  @AT2(-1.0,1.0)  ;ATAN = -0.7853982
```

@ATN(*expression****)***

Returns the arc tangent of *expression* as a floating point value in the range –pi/2 to pi/2. The memory space attribute of the result will be **N**one.

Example:

```
MOVE  #@FRC(@ATN(1.0)),A   ;load arc tangent
```

@CEL(*expression****)***

Returns a floating point value which represents the smallest integer greater than or equal to *expression*. The memory space attribute of the result will be **N**one.

Example:

```
CEIL  SET  @CEL(-1.05)     ;CEIL = -1.0
```

@CNT()

Returns the count of the current macro expansion arguments as an integer. A warning will be issued if this function is used when no macro expansion is active. The memory space attribute of the result will be **N**one.

Example:

```
ARGCNT  SET  @CNT()        ;squirrel away arg count
```

@COH(**expression**)

Returns the hyperbolic cosine of *expression* as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
HYCOS  EQU  @COH(VAL)       ;compute hyperbolic cosine
```

@COS(**expression**)

Returns the cosine of *expression* as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
DC  –@COS(@CVF(COUNT)*FREQ) ;compute cosine value
```

@CVF(**expression**)

Converts the result of *expression* to a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
FLOAT  SET@CVF(5)         ;FLOAT = 5.0
```

@CVI(**expression**)

Converts the result of *expression* to an integer value. This function should be used with caution since the conversions can be inexact (e.g., floating point values are truncated). The memory space attribute of the result will be **N**one.

Example:

```
INT  SET  @CVI(–1.05)      ;INT = –1
```

@CVS(**{**X | Y | L | P | E | N**},expression**)

Converts the memory space attribute of *expression* to that specified by the first argument; returns *expression*. See section *Memory Spaces* for more information on memory space attributes. The *expression* may be relative or absolute.

Example:

```
LOADDR EQU  @CVS(X,TARGET)  ;set LOADDR to X:TARGET
```

@DEF(symbol)

Returns an integer 1 (memory space attribute **N**) if *symbol* has been defined, 0 otherwise. *symbol* may be any label not associated with a **MACRO** or **SECTION** directive. If *symbol* is quoted it is looked up as a **DEFINE** symbol; if it is not quoted it is looked up as an ordinary label.

Example:

```
IF  @DEF(ANGLE)         ;assemble if ANGLE defined
```

@DEFMEM()

You can use the @DEFMEM() function just as the _DEFMEM function in the compiler to retrieve the selected default memory.

Example:

```
IF   @DEFMEM()==’x’
msg  ’default memory x’
ENDIF
IF   @DEFMEM()==’y’
msg  ’default memory y’
ENDIF
IF   @DEFMEM()==’p’
msg  ’default memory p’
ENDIF
IF   @DEFMEM()==’l’
msg  ’default memory l’
ENDIF
```

@DSP()

Returns an integer indicating the type of DSP56xxx processor family. This value is **0** for the DSP5600x, **3** for the DSP563xx family or **6** for the DSP566xx family.

Example:

```
DTYPE  SET  @DSP()      ;DTYPE = 3 for DSP563xx
```

@FLD(*base,value,width[,start]****)***

Shift and mask *value* into *base* for *width* bits beginning at bit *start*. If *start* is omitted, zero (least significant bit) is assumed. All arguments must be positive integers and none may be greater than the target word size. Returns the shifted and masked value with a memory space attribute of **N**one.

Example:

```
SWITCH  EQU  @FLD(TOG,1,1,7)   ;turn eighth bit on
```

@FLR(*expression****)***

Returns a floating point value which represents the largest integer less than or equal to *expression*. The memory space attribute of the result will be **N**one.

Example:

```
FLOOR   SET  @FLR(2.5)         ;FLOOR = 2.0
```

@FRC(*expression****)***

This function performs scaling and convergent rounding to obtain the fractional representation of the floating point *expression* as an integer. The memory space attribute of the result will be **N**one.

Example:

```
FRAC    EQU  @FRC(FLT)+1       ;compute saturation
```

@L10(*expression****)***

Returns the base 10 logarithm of *expression* as a floating point value. *expression* must be greater than zero. The memory space attribute of the result will be **N**one.

Example:

```
LOG     EQU  @L10(100.0)       ;LOG = 2
```

@LEN(*string****)***

Returns the length of *string* as an integer. The memory space attribute of the result will be **N**one.

Example:

```
SLEN   SET  @LEN('string')     ;SLEN = 6
```

@LFR(*expression****)***

This function performs scaling and convergent rounding to obtain the fractional representation of the floating point *expression* as a long integer. The memory space attribute of the result will be **N**one.

Example:

```
LFRAC  EQU  @LFR(LFLT)         ;store binary form
```

@LNG(*expr1,expr2****)***

Concatenates the single word *expr1* and *expr2* into a double word value such that *expr1* is the high word and *expr2* is the low word. The memory space attribute of the result will be **N**one.

Example:

```
LWORD  DC   @LNG(HI,LO)        ;build long word
```

@LOG(*expression****)***

Returns the natural logarithm of *expression* as a floating point value. *expression* must be greater than zero. The memory space attribute of the result will be **N**one.

Example:

```
LOG    EQU  @LOG(100.0)        ;LOG = 4.605170
```

@LST()

Returns the value of the **LIST** directive flag as an integer, with a memory space attribute of **N**one. Whenever a **LIST** directive is encountered in the assembler source, the flag is incremented; when a **NOLIST** directive is encountered, the flag is decremented.

Example:

```
DUP    @CVI(@ABS(@LST()))      ;list unconditionally
```

@LUN(*expression****)***

Converts the double–word *expression* to a floating point value. *expression* should represent a binary fraction. The memory space attribute of the result will be **N**one.

Example:

```
DBLFRC  EQU  @LUN($3FE0000000000000) ;DBLFRC = 0.5
```

@MAC(*symbol****)***

Returns an integer 1 (memory space attribute **N** if *symbol* has been defined as a macro name, 0 otherwise.

Example:

```
IF    @MAC(DOMUL)                ;expand macro
```

@MAX(*expr1[,exprN]...****)***

Returns the greatest of *expr1*,...,*exprN* as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
MAX   DC   @MAX(1.0,5.5,–3.25)   ;MAX = 5.5
```

@MIN(*expr1[,exprN]...****)***

Returns the least of *expr1*,...,*exprN* as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
MIN   DC   @MIN(1.0,5.5,–3.25)    ;Min = –3.25
```

@MODEL()

The @MODEL() function returns the memory model just as the_MODEL function in the compiler. The possible values are derived from the **–M** option.

Example:

```
IF   @MODEL()==24
msg  ’24–bit model’
ENDIF
```

```
IF   @MODEL()==16
msg  ’16-bit model’
ENDIF

IF   @MODEL()==1624
msg  ’16/24-bit model’
ENDIF

IF   @MODEL()==6
msg  ’DSP566xx model’
ENDIF
```

@MSP(**expression**)

Returns the memory space attribute of *expression* as an integer value.

None	= 0
X space	= 1
Y space	= 2
L space	= 3
P space	= 4
E space	= 5

The *expression* may be relative or absolute.

Example:

```
MEM   SET  @MSP(ORIGIN)      ;save memory space
```

@MXP()

Returns an integer 1 (memory space attribute **N**) if the assembler is expanding a macro, 0 otherwise.

Example:

```
IF    @MXP()                 ;macro expansion active?
```

@POS(*str1,str2[,start]* ***)***

Returns the position *str2* in *str1* as an integer, starting at position *start*. If *start* is not given the search begins at the beginning of *str1*. If the *start* argument is specified it must be a positive integer and cannot exceed the length of the source string. The memory space attribute of the result will be **N**one.

Example:

```
ID    EQU  @POS('DSP56000','56')   ;ID = 3
```

@POW(*expr1,expr2* ***)***

Returns *expr1* raised to the power *expr2* as a floating point value. *expr1* and *expr2* must be separated by a comma. The memory space attribute of the result will be **N**one.

Example:

```
BUF   EQU  @CVI(@POW(2.0,3.0))     ;BUF = 8
```

@RND()

Returns a random value in the range 0.0 to 1.0. The memory space attribute of the result will be **N**one.

Example:

```
SEED  DC   @RND()           ;save initial SEED value
```

@RVB(*expr1,expr2* ***)***

Reverse the bits in *expr1* delimited by the number of bits in *expr2*. If *expr2* is omitted the field is bounded by the target word size. Both expressions must be single word integer values.

Example:

```
REV   EQU  @RVB(VAL)        ;reverse all bits in value
```

@SCP(*str1,str2* ***)***

Returns an integer 1 (memory space attribute **N**) if the two strings compare, 0 otherwise. The two strings must be separated by a comma.

Example:

```
IF    @SCP(STR,'MAIN')      ;does STR equal MAIN?
```

@SGN(**expression**)

Returns the sign of *expression* as an integer: –1 if the argument is negative, 0 if zero, 1 if positive. The memory space attribute of the result will be **N**one. The *expression* may be relative or absolute.

Example:

```
IF    @SGN(INPUT)              ;is sign positive?
```

@SIN(**expression**)

Returns the sine of *expression* as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
DC    @SIN(@CVF(COUNT)*FREQ)   ;compute sine value
```

@SNH(**expression**)

Returns the hyperbolic sine of *expression* as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
HSINE EQU  @SNH(VAL)           ;hyperbolic sine
```

@SQT(**expression**)

Returns the square root of *expression* as a floating point value. *expression* must be positive. The memory space attribute of the result will be **N**one.

Example:

```
SQRT  EQU  @SQT(3.5)           ;SQRT = 1.870829
```

@STKMEM()

You can use the @STKMEM() function just as the _STKMEM function in the compiler to retrieve the selected stack memory. @STKMEM only differs from @DEFMEM if the stack is placed in L memory for X or Y default memory.

Example:

```
IF   @STKMEM()==’x’
msg  ’stack memory x’
ENDIF
IF   @STKMEM()==’y’
msg  ’stack memory y’
ENDIF
IF   @STKMEM()==’p’
msg  ’stack memory p’
ENDIF
IF   @STKMEM()==’l’
msg  ’stack memory l’
ENDIF
```

@TAN(expression)

Returns the tangent of *expression* as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
MOVE  #@TAN(1.0),A             ;load tangent
```

@TNH(expression)

Returns the hyperbolic tangent of *expression* as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
HTAN  =  @TNH(VAL)             ;hyperbolic tangent
```

@UNF(expression)

Converts *expression* to a floating point value. *expression* should represent a binary fraction. The memory space attribute of the result will be **N**one.

Example:

```
FRC   EQU  @UNF($400000)       ;FRC = 0.5
```

*@**XPN(**expression**)***

Returns the exponential function (base e raised to the power of *expression*) as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
EXP   EQU  @XPN(1.0)           ;EXP = 2.718282
```

CHAPTER 6

MACRO OPERATIONS

TASKING

CHAPTER 6

6.1 INTRODUCTION

This chapter describes the macro operations and conditional assembly.

The macro preprocessor is implemented in the assembler.

6.2 MACRO OPERATIONS

Programming applications frequently involve the coding of a repeated pattern or group of instructions. Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly for a given occurrence of the instruction group. In either case, macros provide a shorthand notation for handling these instruction patterns. Having determined the iterated pattern, the programmer can, within the macro, designate selected fields of any statement as variable. Thereafter by invoking a macro the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

When the pattern is defined it is given a name. This name becomes the mnemonic by which the macro is subsequently invoked (called). If the name of the macro is the same as an existing assembler directive or mnemonic opcode, the macro will replace the directive or mnemonic opcode, and a warning will be issued.

The macro call causes source statements to be generated. The generated statements may contain substitutable arguments. The statements produced by a macro call are relatively unrestricted as to type. They can be any processor instruction, almost any assembler directive, or any previously–defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions that are applied to statements written by the programmer.

To invoke a macro, the macro name must appear in the operation code field of a source statement. Any arguments are placed in the operand field. By suitably selecting the arguments in relation to their use as indicated by the macro definition, the programmer causes the assembler to produce in–line coding variations of the macro definition.

The effect of a macro call is to produce in–line code to perform a predefined function. The code is inserted in the normal flow of the program so that the generated instructions are executed with the rest of the program each time the macro is called.

An important feature in defining a macro is the use of macro calls within the macro definition. The assembler processes such **nested** macro calls at expansion time only. The nesting of one macro definition within another definition is permitted. However, the nested macro definition will not be processed until the primary macro is expanded. The macro must be defined before its appearance in a source statement operation field.

6.3 MACRO DEFINITION

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the **MACRO** directive, its name, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the **ENDM** directive.

The header of a macro definition has the form:

macro_name **MACRO** [*dummy argument list*] [*comment*]

The required name is the symbol by which the macro will be called. The dummy argument list has the form:

[*dumarg*[**,***dumarg*]...]

The dummy arguments are symbolic names that the macro processor will replace with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as global symbol names. Dummy argument names that are preceded by an underscore are not allowed. Dummy arguments are separated by commas.

When a macro call is executed, the dummy arguments within the macro definition (`NMUL,AVEC,BVEC,OFFSET,RESULT` in the example below) are replaced with the corresponding argument as defined by the macro call.

All local label definitions within a macro are made unique for this macro call (unless the macro label override operator is used, see below). This is done by appending a unique postfix to every local label and removing the leading '_', making the scope of the label local to the module. This mechanism allows the programmer to freely use local labels within a macro definition without regard to the number of times that the macro is expanded.

Non–local labels are considered to be normal labels and thus cannot occur more than once unless used with the **SET** directive (see Chapter 7, *Assembler Directives*).

It is sometimes desirable to pass local labels as macro arguments to be used within the macro as address references (e.g. `MOVE #_LABEL,R0`). The assembler effectively disallows this, however, since underscore label references within a macro invocation are regarded as labels local to that expansion of the macro. If the macro label override operator (**^**) precedes an underscore label the label will not be made unique by appending the unique postfix and removing the leading '_'.

Example

The macro:

```
N_R_MUL   MACRO  NMUL,AVEC,BVEC,OFFSET,RESULT   ;header
    MOVE  #AVEC,R0                              ;body
    MOVE  #BVEC,R4
    MOVE  #OFFSET+^RESULT,R1
    MOVE           X:(R0)+,X0  Y:(R4)+,Y0
    DO    #NMUL,_ENDLOOP
    MPY   Y0,X0,A  X:(R0)+,X0  Y:(R4)+,Y0
    MOVE  A,X:(R1)+
_ENDLOOP
    ENDM                                    ;terminator
```

Invoking this macro with:

```
    N_R_MUL $10,$100,$100,$10,_RESULT
```

expands to: (note the different handling of `_ENDLOOP` and `_RESULT`)

```
    MOVE  #$100,R0
    MOVE  #$100,R4
    MOVE  #$10+_RESULT,R1
    MOVE           X:(R0)+,X0  Y:(R4)+,Y0
    DO    #$10,ENDLOOP_M_Z000001
    MPY   Y0,X0,A  X:(R0)+,X0  Y:(R4)+,Y0
    MOVE  A,X:(R1)+
ENDLOOP_M_Z000001
```

6.4 MACRO CALLS

When a macro is invoked the statement causing the action is termed a **macro call**. The syntax of a macro call consists of the following fields:

[*label*] *macro_name* [*arguments*] [*comment*]

The argument field can have the form:

[*arg*[**,***arg*]...]

The macro call statement is made up of three fields besides the comment field: the *label*, if any, will correspond to the value of the location counter at the start of the macro expansion; the operation field which contains the macro name; and the operand field which contains substitutable arguments. Within the operand field each calling argument of a macro call corresponds one–to–one with a dummy argument of the macro definition. For example, the `N_R_MUL` macro defined earlier could be invoked for expansion (called) by the statement:

```
N_R_MUL   CNT+1,VEC1,VEC2,BASE,OUT
```

where the operand field arguments, separated by commas and taken left to right, correspond to the dummy arguments ”NMUL” through ”RESULT”, respectively. These arguments are then substituted in their corresponding positions of the definition to produce a sequence of instructions.

Macro arguments consist of sequences of characters separated by commas. Although these can be specified as quoted strings, to simplify coding the assembler does not require single quotes around macro argument strings. However, if an argument has an embedded comma or space, that argument must be surrounded by single quotes (**'**). An argument can be declared null when calling a macro. However, it must be declared explicitly null. Null arguments can be specified in four ways:

– by writing the delimiting commas in succession with no intervening spaces;
– by terminating the argument list with a comma and omitting the rest of the argument list;
– by declaring the argument as a null string;
– by simply omitting some or all of the arguments.

A null argument will cause no character to be substituted in the generated statements that reference the argument. If more arguments are supplied in the macro call than appear in the macro definition, a warning will be issued by the assembler.

6.5 DUMMY ARGUMENT OPERATORS

The assembler macro processor provides for text substitution of arguments during macro expansion. In order to make the argument substitution facility more flexible, the assembler also recognizes certain text operators within macro definitions which allow for transformations of the argument text. These operators can be used for text concatenation, numeric conversion, and string handling.

6.5.1 DUMMY ARGUMENT CONCATENATION OPERATOR - \

Dummy arguments that are intended to be concatenated with other characters must be preceded by the concatenation operator, '\' to separate them from the rest of the characters. The argument may precede or follow the adjoining text, but there must be no intervening blanks between the concatenation operator and the rest of the characters. To position an argument between two alphanumeric characters, place a backslash both before and after the argument name. For example, consider the following macro definition:

```
SWAP_REG   MACRO  REG1,REG2  ;swap REG1,REG2
     MOVE  R\REG1,X0         ;using X0 as temp
     MOVE  R\REG2,R\REG1
     MOVE  X0,R\REG2
     ENDM
```

If this macro were called with the following statement,

```
     SWAP_REG  0,1
```

then for the macro expansion, the macro processor would substitute the character 0 for the dummy argument REG1, and the character 1 for the dummy argument REG2. The concatenation operator (\) indicates to the macro processor that the substitution characters for the dummy arguments are to be concatenated in both cases with the character R. The resulting expansion of this macro call would be:

```
MOVE  R0,X0
MOVE  R1,R0
MOVE  X0,R1
```

6.5.2 RETURN VALUE OPERATOR - ?

Another macro definition operator is the question mark (**?**) that returns the value of a symbol. When the macro processor encounters this operator, the **?***symbol* sequence is converted to a character string representing the decimal value of the *symbol*. For example, consider the following modification of the SWAP_REG macro described above:

```
SWAP_SYM   MACRO  REG1,REG2  ;swap REG1,REG2
     MOVE  R\?REG1,X0        ;using X0 as temp
     MOVE  R\?REG2,R\?REG1
     MOVE  X0,R\?REG2
     ENDM
```

If the source file contained the following SET statements and macro call,

```
AREG SET       0
BREG SET       1
     SWAP_SYM  AREG,BREG
```

then the sequence of events would be as follows: the macro processor would first substitute the characters AREG for each occurrence of REG1 and BREG for each occurrence of REG2. For discussion purposes (this would never appear on the source listing), the intermediate macro expansion would be:

```
MOVE  R\?AREG,X0
MOVE  R\?BREG,R\?AREG
MOVE  X0,R\?BREG
```

The macro processor would then replace **?**AREG with the character 0 and **?**BREG with the character 1, since 0 is the value of the symbol AREG and 1 is the value of BREG. The resulting intermediate expansion would be:

```
MOVE  R\0,X0
MOVE  R\1,R\0
MOVE  X0,R\1
```

Next, the macro processor would apply the concatenation operator (\), and the resulting expansion as it would appear on the source listing would be:

```
MOVE  R0,X0
MOVE  R1,R0
MOVE  X0,R1
```

6.5.3 RETURN HEX VALUE OPERATOR - %

The percent sign (**%**) is similar to the standard return value operator except that it returns the hexadecimal value of a symbol. When the macro processor encounters this operator, the **%***symbol* sequence is converted to a character string representing the hexadecimal value of the *symbol*. Consider the following macro definition:

```
GEN_LAB   MACRO  LAB,VAL,STMT
LAB\%VAL  STMT
     ENDM
```

This macro generates a label consisting of the concatenation of the label prefix argument and a value that is interpreted as hexadecimal. If this macro were called as follows,

```
NUM  SET       10
     GEN_LAB   HEX,NUM,'NOP'
```

the macro processor would first substitute the characters HEX for LAB, then it would replace **%**VAL with the character A, since A is the hexadecimal representation for the decimal integer 10. Next, the macro processor would apply the concatenation operator (\). Finally, the string 'NOP' would be substituted for the STMT argument. The resulting expansion as it would appear in the listing file would be:

```
HEXA NOP
```

The percent sign is also the character used to indicate a binary constant. If a binary constant is required inside a macro it may be necessary to enclose the constant in parentheses or escape the constant by following the percent sign by a backslash (\).

6.5.4 DUMMY ARGUMENT STRING OPERATOR - ”

Another dummy argument operator is the double quote (”). This character is replaced with a single quote by the macro processor, but following characters are still examined for dummy argument names. The effect in the macro call is to transform any enclosed dummy arguments into literal strings. For example, consider the following macro definition:

```
STR_MAC    MACRO  STRING
    DC     ”STRING”
    ENDM
```

If this macro were called with the following macro expansion line,

```
    STR_MAC  ABCD
```

then the resulting macro expansion would be:

```
    DC       ’ABCD’
```

Double quotes also make possible **DEFINE** directive expansion within quoted strings. Because of this overloading of the double quotes, care must be taken to insure against inappropriate expansions in macro definitions. Since **DEFINE** expansion occurs before macro substitution, any **DEFINE** symbols are replaced first within a macro dummy argument string:

```
    DEFINE LONG  ’short’
STR_MAC    MACRO  STRING
    MSG    ’This is a LONG STRING’
    MSG    ”This is a LONG STRING”
    ENDM
```

If this macro were invoked as follows,

```
    STR_MAC  sentence
```

then the resulting expansion would be:

```
MSG   'This is a LONG STRING'
MSG   'This is a short sentence'
```

6.5.5 MACRO LOCAL LABEL OVERRIDE OPERATOR - ^

It may be desirable to pass a local label as a macro argument to be used as an address reference within the macro body. If a circumflex (**^**) precedes an underscore operator than the macro preprocessor will not perform any name mangling on that label so the label is used literally in the resulting macro expansion. Here is an example:

```
LOAD  MACRO  ADDR
      MOVE   P:^ADDR,R0
      ENDM
```

The macro ^–operator prevents name mangling on the ADDR argument if the argument has a leading underscore. If there is no leading underscore on the actual argument the the ^–operator has no effct. Consider the following macro call:

```
_LOCAL  LOAD  _LOCAL
```

Without the local label override in the macro definition the macro LOAD would expand to the something like this:

```
_LOCAL
        MOVE  P:LOCAL_M_Z000001,R0
```

This would result in an assembly error as the label `LOCAL_M_Z000001` is nowhere defined. With the local label override in the macro definition (as shown above) the macro LOAD would expand, as expected, to this:

```
_LOCAL
        MOVE  P:_LOCAL,R0
```

This will assemble correctly.

After macro expansion the normal scoping rules on local labels still apply. So, when the following macro is defined:

```
ERROR MACRO ADDR
_LABEL
      MOVE  P:^ADDR,R0
      ENDM
```

And that macro is used in the following way:

```
_LABEL
      MOVE  R0,A
      ERROR _LOCAL
```

The resulting program, after macro expansion, would be something like this:

```
_LOCAL
      MOVE  R0,A
LABEL_M_Z000002:
      MOVE  P:_LOCAL,R0
```

Which is an incorrect program as the name mangled macro label hides the definition of _LOCAL to the second MOVE statement. This results in an assembly error.

6.6 DUP, DUPA, DUPC, DUPF DIRECTIVES

The **DUP**, **DUPA**, **DUPC**, and **DUPF** directives are specialized macro forms. They can be thought of as a simultaneous definition and call of an unnamed macro. The source statements between the **DUP**, **DUPA**, **DUPC**, and **DUPF** directives and the **ENDM** directive follow the same rules as macro definitions, including (int the case of **DUPA**, **DUPC**, and **DUPF**) the dummy operator characters described previously.

For a detailed description of these directives, refer to Chapter 7, *Assembler Directives*.

6.7 CONDITIONAL ASSEMBLY

Conditional assembly facilitates the writing of comprehensive source programs that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros, and through definition of symbols via the **DEFINE**, **SET**, and **EQU** directives. Variations of parameters can then cause assembly of only those parts necessary for the given conditions. The built–in functions of the assembler provide a versatile means of testing many conditions of the assembly environment

See section 5.4 for more information on the assembler built–in functions.

Conditional directives can also be used within a macro definition to ensure at expansion time that arguments fall within a range of allowable values. In this way macros become self–checking and can generate error messages to any desired level of detail.

The conditional assembly directive **IF** has the following form:

```
IF   expression
 .
 .
[ELSE]     ;(the ELSE directive is optional)
 .
 .
ENDIF
```

A section of a program that is to be conditionally assembled must be bounded by an **IF–ENDIF** directive pair. If the optional **ELSE** directive is not present, then the source statements following the **IF** directive and up to the next **ENDIF** directive will be included as part of the source file being assembled only if the *expression* had a nonzero result. If the *expression* has a value of zero, the source file will be assembled as if those statements between the **IF** and the **ENDIF** directives were never encountered. If the **ELSE** directive is present and *expression* has a nonzero result, then the statements between the **IF** and **ELSE** directives will be assembled, and the statement between the **ELSE** and **ENDIF** directives will be skipped. Alternatively, if *expression* has a value of zero, then the statements between the **IF** and **ELSE** directives will be skipped, and the statements between the **ELSE** and **ENDIF** directives will be assembled.

CHAPTER 7

ASSEMBLER DIRECTIVES

TASKING

CHAPTER 7

7.1 OVERVIEW

Assembler directives, or pseudo instructions, are used to control the assembly process. Rather than being translated into a DSP56xxx machine instruction, assembler directives are interpreted by the assembler. The directives perform actions such as assembly control, listing control, defining symbols or changing the location counter. Upper and lower case letters are considered equivalent for assembler directives.

There are some implementation differences between the TASKING DSP56xxx assembler and the Motorola CLAS assembler as described in the Appendix *Migration from Motorola CLAS*. The following directives are new:

ALIGN	–	Specify alignment
EXTERN	–	Declare extern symbols
SYMB	–	Pass high–level language debug information to the object file
CALLS	–	Pass call information to object file. Used to build a call tree at link time for overlaying overlay sections.
VOID	–	Control DO loop optimization.

Assembler directives can be grouped by function into eight types:

1. Debugging
2. Assembly control
3. Symbol definition
4. Data definition/storage allocation
5. Listing control and options
6. Object file control
7. Macros and conditional assembly
8. Structured programming

7.1.1 DEBUGGING

The compiler generates the following directives to pass high level language symbolic debug information via the assembler into the object file:

CALLS	–	Pass call information to object file. Used to build a call tree at link time for overlaying overlay sections.
SYMB	–	Pass symbolic debug information

7.1.2 ASSEMBLY CONTROL

The directives used for assembly control are:

ALIGN	–	Specify alignment
COMMENT	–	Start comment lines. This directive is not permitted in IF/ELSE/ENDIF constructs and MACRO/DUP definitions.
DEFINE	–	Define substitution string
END	–	End of source program
FAIL	–	Programmer generated error message
FORCE	–	Set operand forcing mode
HIMEM	–	Accepted for compatibility and ignored
INCLUDE	–	Include secondary file
LOMEM	–	Accepted for compatibility and ignored
MODE	–	Accepted for compatibility. A warning is issued when it is used.
MSG	–	Programmer generated message
ORG	–	Initialize memory space and location counters. The syntax of the ORG directive is extended for type checking. The overlay specification part of the ORG directive is accepted for compatibility. When it is supplied the assembler issues a warning.
RADIX	–	Change input radix for constants
RDIRECT	–	Accepted for compatibility and ignored
SCSJMP	–	Set structured control branching mode

SCSREG	–	Reassign structured control statement register
UNDEF	–	Undefine **DEFINE** symbol
VOID	–	Control DO loop optimization
WARN	–	Programmer generated warning

7.1.3 SYMBOL DEFINITION

The directives used to control symbol definition are:

ENDSEC	–	End section
EQU	–	Equate symbol to a value; accepts forward references
EXTERN	–	External symbol declaration; also permitted in module body
GLOBAL	–	Global symbol declaration; also permitted in module body
GSET	–	Set global symbol to a value; accepts forward references
LOCAL	–	Local symbol declaration
SECTION	–	Start section. Section scoping is resolved on module basis by the assembler and not on program basis by the linker. The results may differ from those with the Motorola CLAS assembler.
SET	–	Set symbol to a value; accepts forward references
XDEF	–	The assembler treats this directive as GLOBAL directive. A warning is issued when the XDEF directive is used.
XREF	–	The assembler treats this directive as EXTERN directive. A warning is issued when the XREF directive is used.

7.1.4 DATA DEFINITION/STORAGE ALLOCATION

The directives used to control constant data definition and storage allocation are:

BADDR	–	Set buffer address
BSB	–	Block storage bit–reverse
BSC	–	Block storage of constant

BSM	–	Block storage modulo
BUFFER	–	Start buffer
DC	–	Define constant
DCB	–	Define constant byte
DS	–	Define storage
DSM	–	Define modulo storage
DSR	–	Define reverse carry storage
ENDBUF	–	End buffer

7.1.5 LISTING CONTROL AND OPTIONS

The directives used to control the output listing are:

LIST	–	List the assembly
LSTCOL	–	Accepted for compatibility and ignored
NOLIST	–	Stop assembly listing
OPT	–	Assembler options. Not all options of the OPT directive are supported due to the different architecture of the TASKING assembler. Options that are not recognized are ignored.
PAGE	–	Top of page/size page
PRCTL	–	Send control string to printer
STITLE	–	Initialize program subtitle
TABS	–	Set listing tab stops
TITLE	–	Initialize program title

7.1.6 OBJECT FILE CONTROL

The following object file control directives are not supported. A warning is issued when the assembler encounters them in the input file.

COBJ	–	Accepted for compatibility
IDENT	–	Accepted for compatibility

SYMOBJ	–	Accepted for compatibility

7.1.7 MACROS AND CONDITIONAL ASSEMBLY

The directives used for macros and conditional assembly are:

DUP	–	Duplicate sequence of source lines
DUPA	–	Duplicate sequence with arguments
DUPC	–	Duplicate sequence with characters
DUPF	–	Duplicate sequence in loop
ENDIF	–	End of conditional assembly
ENDM	–	End of macro definition
EXITM	–	Exit macro
IF	–	Conditional assembly directive
MACLIB	–	Accepted for compatibility and ignored
MACRO	–	Macro definition
PMACRO	–	Purge macro definition

7.1.8 STRUCTURED PROGRAMMING

The directives used for structured programming are:

.BREAK	–	Exit from structured loop construct
.CONTINUE	–	Continue next iteration of structured loop
.ELSE	–	Perform following statements when .IF false
.ENDF	–	End of .FOR loop
.ENDI	–	End of .IF condition
.ENDL	–	End of hardware loop
.ENDW	–	End of .WHILE loop
.FOR	–	Begin .FOR loop
.IF	–	Begin .IF condition
.LOOP	–	Begin hardware loop

.REPEAT	–	Begin .REPEAT loop
.UNTIL	–	End of .REPEAT loop
.WHILE	–	Begin .WHILE loop

7.2 DIRECTIVES

The rest of this chapter contains individual descriptions of each of the assembler directives, listed alphabetically. They include usage guidelines, functional descriptions, and examples. Some directives require a label field, while in many cases a label is optional. If the description of an assembler directive does not indicate a mandatory or optional label field, then a label is not allowed on the same line as the directive.

Structured programming directives are discussed separately in Chapter 8, *Structured Control Statements*.

ALIGN

Syntax:

ALIGN *expression*

ALIGN CACHE (**as563** only)

Description:

Align the location counter. The *expression* must be represented by a value of 2^k. The default alignment is on a multiple of 1 word. *expression* must be greater than 0. If *expression* is not a value of 2^k, a warning is issued and the alignment will be set to the next 2^k value. Alignment will be performed once at the place where you write the align pseudo. The start of a section is aligned automatically to the largest alignment value occurring in that section.

With **ALIGN CACHE** the **as563** assembler aligns the current location on a code cache boundary. The assembler inserts a gap at the start of the section so that the current location is aligned on a cache boundary. This alignment can only be done once per section.

Modulo– and reverse–carry buffers must be aligned on an address that equals to the first power of 2 greater than or equal to the given buffer expression. For example:

```
Directive      Aligns on

dsm  $20       $20   (2^5)
dsr  $102      $200  (2^9)
```

Depending on the section type the assembler has two cases for these directives:

– Relocatable sections
  The section will be aligned on the calculated alignment boundary. A gap is generated depending on the current relative location counter for this section.
– Absolute sections
  The section location is not changed.
  A gap is generated according to the current absolute address.

Examples:

```
ALIGN 4         ;align at 4 words
lab1: ALIGN 6   ;not a 2^k value.
                ;a warning is issued
                ;lab1 is aligned on 8 words
```

DSP563xx only:

```
ALIGN CACHE     ;align on code cache boundary
```

BADDR

Syntax:

BADDR {**M** | **R**},*expression*

Description:

Set buffer address. The **BADDR** directive sets the run–time location counter to the address of a buffer of the give type, the length of which in words is equal to the value of *expression*. The buffer type may be either **M**odulo or **R**everse–carry. If the run–time location counter is not zero, this directive first advances the run–time location counter to a base address that is a multiple of 2^k, where 2^k>=*expression*. An error will be issued if there is insufficient memory remaining to establish a valid base address. Unlike other buffer allocation directives, the run–time location counter is **<u>not</u>** advanced by the value of the integer expression in the operand field; the location counter remains at the buffer base address. The block of memory intended for the buffer is not initialized to any value.

The result of *expression* may have any memory space attribute but must be an absolute integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined). If a **M**odulo buffer is specified, the expression must fall within the range 2 <= *expression* <= *m*, where *m* is the maximum address of the target DSP. If a **R**everse–carry buffer is designated and *expression* is not a power of two a warning will be issued.

A label is not allowed with this directive.

Examples:

```
      ORG X:$100
      BADDR M,24  ;Circular buffer MOD 24
M_BUF DS  24
```

BSM, **BSB**, **BUFFER**, **DSM**, **DSR**

BSB

Syntax:

[*label*] **BSB** *expression*[**,***expression*]

Description:

Block Storage Bit–Reverse. The **BSB** directive causes the assembler to allocate and initialize a block of words for a reverse–carry buffer. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the run–time location counter is not zero, this directive first advances the run–time location counter to a base address that is a multiple of 2^k, where 2^k is greater than or equal to the value of the first expression. An error will occur if the first expression contains address labels that are not yet defined (forward references) or if the expression has a value of less than or equal to zero. Also, if the first expression is not a power of two a warning will be generated. Both expressions can have any memory space attribute.

label, if present, will be assigned the value of the run–time location counter after a valid base address has been established.

Only one word of object code will be shown on the listing, regardless of how large the first expression is. However, the run–time location counter will be advanced by the number of words generated.

Examples:

```
BUFFER  BSB  BUFSIZ     ;Initialize buffer to zeros
```

BSC, **BSM**, **DC**

BSC

Syntax:

[*label*] **BSC** *expression*[,*expression*]

Description:

Block Storage of Constant. The **BSC** directive causes the assembler to allocate and initialize a block of words. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the first expression contains address labels that are not yet defined (forward references) or if the expression has a value of less than or equal to zero, an error will be generated. Both expressions can have any memory space attribute.

label, if present, will be assigned the value of the run–time location counter at the start of the directive processing.

Only one word of object code will be shown on the listing, regardless of how large the first expression is. However, the run–time location counter will be advanced by the number of words generated.

Examples:

```
TOP    equ $0F
BOTTOM equ $0A
aLabel BSC  TOP-BOTTOM,$055
```

BSB, **BSM**, **DC**

BSM

Syntax:

[*label*] **BSM** *expression*[,*expression*]

Description:

Block Storage Modulo. The **BSM** directive causes the assembler to allocate and initialize a block of words for a modulo buffer. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the run–time location counter is not zero, this directive first advances the run–time location counter to a base address that is a multiple of 2^k, where 2^k is greater than or equal to the value of the first expression. An error will occur if the first expression contains address labels that are not yet defined (forward references), has a value of less than or equal to zero, or falls outside the range 2 <= *expression* <= *m*, where *m* is the maximum address of the target DSP. Both expressions can have any memory space attribute.

label, if present, will be assigned the value of the run–time location counter after a valid base address has been established.

Only one word of object code will be shown on the listing, regardless of how large the first expression is. However, the run–time location counter will be advanced by the number of words generated.

Examples:

```
BUFFER  BSM  BUFSIZ,$FFFFFFFF
        ;Initialize buffer to all ones
```

BSB, **BSC**, **DC**

BUFFER

Syntax:

BUFFER {**M** | **R**},*expression*

Description:

Start Buffer. The **BUFFER** directive indicates the start of a buffer of the given type. Data is allocated for the buffer until an **ENDBUF** directive is encountered. Instructions and most data definition directives may appear between the **BUFFER** and **ENDBUF** pair, although **BUFFER** directives may not be nested and certain types of directives such as **ORG**, **SECTION**, and other buffer allocation directives may not be used. The *expression* represents the buffer size. If less data is allocated than the size of the buffer, the remaining buffer locations will be uninitialized. If more data is allocated than the specified size of the buffer, an error is issued.

The **BUFFER** directive sets the run–time location counter to the address of a buffer of the given type, the length of which in words is equal to the value of *expression*. The buffer type may be either **M**odulo or **R**everse–carry. If the run–time location counter is not zero, this directive first advances the run–time location counter to a base address that is a multiple of 2^k, where $2^k >=$ *expression*. An error will be issued if there is insufficient memory remaining to establish a valid base address. Unlike other buffer allocation directives, the run–time location counter is **not** advanced by the value of the integer expression in the operand field; the location counter remains at the buffer base address.

The result of *expression* may have any memory space attribute but must be an absolute integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined). If a **M**odulo buffer is specified, the expression must fall within the range 2 <= *expression* <= *m*, where *m* is the maximum address of the target DSP. If a **R**everse–carry buffer is designated and *expression* is not a power of two a warning will be issued.

For example:

```
Directive     Aligns on

dsm  $20      $20   (2^5)
dsr  $102     $200  (2^9)
```

Depending on the section type the assembler has two cases for these directives.

- Relocatable sections
  The section will be aligned on the calculated alignment boundary. A gap is generated depending on the current relative location counter for this section.
- Absolute sections
  The section location is not changed.
  A gap is generated according to the current absolute address.

A label is not allowed with this directive.

Examples:

```
      ORG  X:$100
      BUFFER M,24  ;Circular buffer MOD 24
M_BUF DC   0.5,0.5,0.5,0.5
      DS  20       ;Remainder uninitialized
      ENDBUF
```

BADDR, **BSM**, **BSB**, **DSM**, **DSR**, **ENDBUF**

CALLS

Syntax:

CALLS ’*caller*’**,** ’*callee*’ [**,** ’*callee*’]...

Description:

Create a flow graph reference between *caller* and *callees*. The linker needs this information to build a flow graph, which steers the overlay algorithm. *caller* and *callee* are names of functions.

The CALLS directive should be used in hand coded assembly when the assembly code calls a C function. Make sure that the hand coded CALLS directive connects to the compiler generated call graph, i.e. the name of the caller must also be named as a callee in another CALLS directive.

Examples:

```
CALLS ’main’, ’nfunc’
```

COMMENT

Syntax:

COMMENT *delimiter*
.
.
delimiter

Description:

Start Comment Lines. The **COMMENT** directive is used to define one or more lines as comments. The first non–blank character after the **COMMENT** directive is the comment delimiter. The two delimiters are used to define the comment text. The line containing the second comment delimiter will be considered the last line of the comment. The comment text can include any printable characters and the comment text will be produced in the source listing as it appears in the source file.

A label is not allowed with this directive.

This directive is not permitted in IF/ELSE/ENDIF constructs and MACRO/DUP definitions.

Examples:

```
COMMENT  + This is a one line comment +
COMMENT  * This is a multiple line
           comment. Any number of lines
           can be placed between the two
           delimiters.
         *
```

DC

Syntax:

[*label*] **DC** *arg*[**,***arg*]...

Description:

Define Constant. The **DC** directive allocates and initializes a word of memory for each *arg* argument. *arg* may be a numeric constant, a single or multiple character string constant, a symbol, or an expression. The **DC** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. If the **DC** directive is used in L memory, the arguments will be evaluated and stored as long word quantities. Otherwise, an error will occur if the evaluated argument value is too large to represent in a single DSP word.

label, if present, will be assigned the value of the run–time location counter at the start of the directive processing.

Integer arguments are stored as is; floating point numbers are converted to binary values. Single and multiple character strings are handled in the following manner:

1. Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character.

   Example: `’R’ = $000052`

2. Multiple character strings represent words whose bytes are composed of concatenated sequences of the ASCII representation of the characters in the string (unless the **NOPS** option is specified; see the **OPT** directive). If the number of characters is not an even multiple of the number of bytes per DSP word, then the last word will have the remaining characters left aligned and the rest of the word will be zero–filled. If the **NOPS** option is given, each character in the string is stored in a word whose lower seven bits represent the ASCII value of the character.

   Example:

```
’ABCD’ = $414243
         $440000
```

Examples:

```
TABLE  DC  1426,253,$2662,'ABCD'
CHARS  DC  'A','B','C','D'
```

BSC, **DCB**

DCB

Syntax:

[*label*] **DCB** *arg*[,*arg*]...

Description:

Define Constant Byte. The **DCB** directive allocates and initializes a byte of memory for each *arg* argument. *arg* may be a byte integer constant, a single or multiple character string constant, a symbol, or a byte expression. The **DCB** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive byte locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding byte location will be filled with zeros.

label, if present, will be assigned the value of the run–time location counter at the start of the directive processing.

Integer arguments are stored as is, but must be byte values (e.g. within the range 0–255); floating point numbers are not allowed. Single and multiple character strings are handled in the following manner:

1. Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character.

   Example: `’R’ = $000052`

2. Multiple character strings represent words whose bytes are composed of concatenated sequences of the ASCII representation of the characters in the string (unless the **NOPS** option is specified; see the **OPT** directive). If the number of characters is not an even multiple of the number of bytes per DSP word, then the last word will have the remaining characters left aligned and the rest of the word will be zero–filled. If the **NOPS** option is given, each character in the string is stored in a word whose lower seven bits represent the ASCII value of the character.

   Example:

```
’AB’,,’CD’ = $414200
             $434400
```

Examples:

```
TABLE  DCB  'two',0,'strings',0
CHARS  DCB  'A','B','C','D'
```

BSC, **DC**

DEFINE

Syntax:

DEFINE *symbol string*

Description:

Define Substitution String. The **DEFINE** directive is used to define substitution strings that will be used on all following source lines. All succeeding lines will be searched for an occurrence of *symbol*, which will be replaced by *string*. This directive is useful for providing better documentation in the source program. *symbol* must adhere to the restrictions for non–local labels. That is, the first of which must be alphabetic, and the remainder of which must be either alphanumeric or the underscore (**_**). A warning will result if a new definition of a previously defined symbol is attempted.

Macros represent a special case. **DEFINE** directive translations will be applied to the macro definition as it is encountered. When the macro is expanded any active **DEFINE** directive translations will again be applied.

DEFINE directive symbols that are defined within a section will only apply to that section. See the **SECTION** directive.

A label is not allowed with this directive.

Examples:

If the following **DEFINE** directive occurred in the first part of the source program:

```
DEFINE  ARRAYSIZ  '10 * SAMPLSIZ'
```

then the source line below:

```
DS      ARRAYSIZ
```

would be transformed by the assembler to the following:

```
DS      10 * SAMPLSIZ
```

UNDEF

DS

Syntax:

[*label*] **DS** *expression*

Description:

Define Storage. The **DS** directive reserves a block of memory the length of which in words is equal to the value of *expression*. This directive causes the run–time location counter to be advanced by the value of the absolute integer expression in the operand field. *expression* can have any memory space attribute. The block of memory reserved is not initialized to any value. The expression must be an integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined).

label, if present, will be assigned the value of the run–time location counter at the start of the directive processing.

Examples:

```
S_BUF  DS  12     ; Sample buffer
```

DSM, **DSR**

DSM

Syntax:

[*label*] **DSM** *expression*

Description:

Define Modulo Storage. The **DSM** directive reserves a block of memory the length of which in words is equal to the value of *expression*. If the run–time location counter is not zero, this directive first advances the run–time location counter to a base address that is a multiple of 2^k, where 2^k >= *expression*. An error will be issued if there is insufficient memory remaining to establish a valid base address. Next the run–time location counter is advanced by the value of the integer expression in the operand field. *expression* can have any memory space attribute. The block of memory reserved is not initialized to any given value. The result of *expression* must be an absolute integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined). The expression also must fall within the range 2 <= *expression* <= *m*, where *m* is the maximum address of the target DSP.

For example:

```
dsm  $20    ; aligns on $20 (2^5)
```

Depending on the section type the assembler has two cases for this directive.

- Relocatable sections
  The section will be aligned on the calculated alignment boundary. A gap is generated depending on the current relative location counter for this section.
- Absolute sections
  The section location is not changed.
  A gap is generated according to the current absolute address.

label, if present, will be assigned the value of the run–time location counter after a valid base address has been established.

Examples:

```
        ORG   X:$100
M_BUF   DSM   24        ; Circular buffer MOD 24
```

DS, **DSR**

DSR

Syntax:

[*label*] **DSR** *expression*

Description:

Define Reverse Carry Storage. The **DSR** directive reserves a block of memory the length of which in words is equal to the value of *expression*. If the run–time location counter is not zero, this directive first advances the run–time location counter to a base address that is a multiple of 2^k, where $2^k >=$ *expression*. An error will be issued if there is insufficient memory remaining to establish a valid base address. Next the run–time location counter is advanced by the value of the integer expression in the operand field. *expression* can have any memory space attribute. The block of memory reserved is not initialized to any given value. The result of *expression* must be an absolute integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined). Since the **DSR** directive is useful mainly for generating FFT buffers, if *expression* is not a power of two a warning will be generated.

For example:

```
dsr  $102    ; aligns on $200  (2^9)
```

Depending on the section type the assembler has two cases for this directive.

- Relocatable sections
  The section will be aligned on the calculated alignment boundary. A gap is generated depending on the current relative location counter for this section.
- Absolute sections
  The section location is not changed.
  A gap is generated according to the current absolute address.

label, if present, will be assigned the value of the run–time location counter after a valid base address has been established.

Examples:

```
       ORG  X:$100
R_BUF  DSR  8      ; Reverse carry buffer for
                   ; 16 point FFT
```

DS, **DSM**

DUP

Syntax:

[*label*] **DUP** *expression*
.
.
ENDM

Description:

Duplicate Sequence of Source Lines. The sequence of source lines between the **DUP** and **ENDM** directives will be duplicated by the number specified by the integer *expression*. *expression* can have any memory space attribute. If the expression evaluates to a number less than or equal to 0, the sequence of lines will not be included in the assembler output. The expression result must be an absolute integer and cannot contain any forward references to address labels (labels that have not already been defined). The **DUP** directive may be nested to any level.

label, if present, will be assigned the value of the run–time location counter at the start of the **DUP** directive processing.

Examples:

The sequence of source input statements,

```
COUNT   SET  3
        DUP  COUNT    ; ASR BY COUNT
        ASR  D0
        ENDM
```

would generate the following in the source listing:

```
COUNT   SET  3
        DUP  COUNT    ; ASR BY COUNT
        ASR  D0
        ASR  D0
        ASR  D0
        ENDM
```

Note that the lines

```
DUP    COUNT         ;ASR BY COUNT
ENDM
```

will only be shown on the source listing if the **MD** option is enabled. The lines

```
ASR  D0
ASR  D0
ASR  D0
```

will only be shown on the source listing if the **MEX** option is enabled.

See the **OPT** directive in this chapter for more information on the **MD** and **MEX** options.

DUPA, **DUPC**, **DUPF**, **ENDM**, **MACRO**

DUPA

Syntax:

```
[label]    DUPA  dummy,arg[,arg]...
           .
           .
           ENDM
```

Description:

Duplicate Sequence With Arguments. The block of source statements defined by the **DUPA** and **ENDM** directives will be repeated for each argument. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding argument string. If the argument string is a null, then the block is repeated with each occurrence of the dummy parameter removed. If an argument includes an embedded blank or other assembler–significant character, it must be enclosed with single quotes.

label, if present, will be assigned the value of the run–time location counter at the start of the **DUPA** directive processing.

Examples:

If the input source file contained the following statements,

```
DUPA  VALUE,12,32,34
DC    VALUE
ENDM
```

then the assembler source listing would show

```
DUPA  VALUE,12,32,34
DC    12
DC    32
DC    34
ENDM
```

Note that the lines

```
DUPA  VALUE,12,32,34
ENDM
```

will only be shown on the source listing if the **MD** option is enabled.

The lines

```
DC  12
DC  32
DC  34
```

will only be shown on the source listing if the **MEX** option is enabled.

See the **OPT** directive in this chapter for more information on the **MD** and **MEX** options.

DUP, **DUPC**, **DUPF**, **ENDM**, **MACRO**

DUPC

Syntax:

```
[label]    DUPC  dummy,string
           .
           .
           ENDM
```

Description:

Duplicate Sequence With Characters. The block of source statements defined by the **DUPC** and **ENDM** directives will be repeated for each character of *string*. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding character in the string. If the string is null, then the block is skipped.

label, if present, will be assigned the value of the run–time location counter at the start of the **DUPC** directive processing.

Examples:

If the input source file contained the following statements,

```
DUPC  VALUE,'123'
DC    VALUE
ENDM
```

then the assembler source listing would show

```
DUPC  VALUE,'123'
DC    1
DC    2
DC    3
ENDM
```

Note that the lines

```
DUPC  VALUE,'123'
ENDM
```

will only be shown on the source listing if the **MD** option is enabled.

The lines

```
DC  1
DC  2
DC  3
```

will only be shown on the source listing if the **MEX** option is enabled.

See the **OPT** directive in this chapter for more information on the **MD** and **MEX** options.

DUP, **DUPA**, **DUPF**, **ENDM**, **MACRO**

DUPF

Syntax:

[*label*] **DUPF** *dummy*,[*start*],*end*[,*increment*]
.
.
ENDM

Description:

Duplicate Sequence In Loop. The block of source statements defined by the **DUPF** and **ENDM** directives will be repeated in general (*end* – *start*) + 1 ties when *increment* is 1. *start* is the starting value for the loop index; *end* represents the final value. *increment* is the increment for the loop index; it defaults to 1 if omitted (as does the *start* value). The *dummy* parameter holds the loop index value and may be used within the body of instructions.

label, if present, will be assigned the value of the run–time location counter at the start of the **DUPF** directive processing.

Examples:

If the input source file contained the following statements,

```
DUPF  NUM,0,7
MOVE  #0,R\NUM
ENDM
```

then the assembler source listing would show

```
DUPF  NUM,0,7
MOVE  #0,R0
MOVE  #0,R1
MOVE  #0,R2
MOVE  #0,R3
MOVE  #0,R4
MOVE  #0,R5
MOVE  #0,R6
MOVE  #0,R7
ENDM
```

Note that the lines

```
DUPF  NUM,0,7
ENDM
```

will only be shown on the source listing if the **MD** option is enabled. The lines

```
MOVE  #0,R0
MOVE  #0,R1
MOVE  #0,R2
MOVE  #0,R3
MOVE  #0,R4
MOVE  #0,R5
MOVE  #0,R6
MOVE  #0,R7
```

will only be shown on the source listing if the **MEX** option is enabled.

See the **OPT** directive in this chapter for more information on the **MD** and **MEX** options.

DUP, **DUPA**, **DUPC**, **ENDM**, **MACRO**

END

Syntax:

END [*expression*]

Description:

End of Source Program. The optional **END** directive indicates that the logical end of the source program has been encountered. The *expression* is only permitted here for compatibility reasons. It is ignored during assembly. The **END** directive cannot be used in a macro expansion.

A label is not allowed with this directive.

Examples:

```
END            ;End of source program
```

ENDBUF

Syntax:

ENDBUF

Description:

End Buffer. The **ENDBUF** directive is used to signify the end of a buffer block. The run–time location counter will remain just beyond the end of the buffer when the **ENDBUF** directive is encountered.

A label is not allowed with this directive.

Examples:

```
     ORG    :$100
BUF  BUFFER  R,64    ;uninitialized
                     ;reverse-carry buffer
     ENDBUF
```

BUFFER

ENDIF

Syntax:

ENDIF

Description:

End Of Conditional Assembly. The **ENDIF** directive is used to signify the end of the current level of conditional assembly. Conditional assembly directives can be nested to any level, but the **ENDIF** directive always refers to the most previous **IF** directive.

A label is not allowed with this directive.

Examples:

```
IF  DEB
DEBUG     ;Enter debug mode
ENDIF
```

IF

ENDM

Syntax:

ENDM

Description:

End of Macro Definition. Every **MACRO**, **DUP**, **DUPA**, and **DUPC** directive must be terminated by an **ENDM** directive.

A label is not allowed with this directive.

Examples:

```
SWAP_SYM  MACRO  REG1,REG2    ;swap REG1,REG2
    MOVE  R\?REG1,X0          ;using X0 as temp
    MOVE  R\?REG2,R\?REG1
    MOVE  X0,R\?REG2
    ENDM
```

DUP, **DUPA**, **DUPC**, **MACRO**

ENDSEC

Syntax:

ENDSEC

Description:

End Section. Every **SECTION** directive must be terminated by an **ENDSEC** directive.

A label is not allowed with this directive.

Examples:

```
        SECTION  COEFF
        ORG      Y:
VALUES  BSC      $100        ;Initialize to zero
        ENDSEC
```

SECTION

EQU

Syntax:

label **EQU** [{**X:** | **Y:** | **L:** | **P:** | **E:**}]*expression*

Description:

Equate Symbol to a Value. The **EQU** directive assigns the value and memory space attribute of *expression* to the symbol *label*. If *expression* has a memory space attribute of **N**one, then it can optionally be preceded by any of the indicated memory space qualifiers to force a memory space attribute. An error will occur if the expression has a memory space attribute other than **N**one and it is different than the forcing memory space attribute. The optional forcing memory space attribute is useful to assign a memory space attribute to an expression that consists only of constants but is intended to refer to a fixed address in a memory space.

The **EQU** directive is one of the directives that assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program (or section, if **SECTION** directives are being used). The *expression* may be relative or absolute, and forward references are allowed.

Examples:

```
A_D_PORT   EQU   X:$4000
```

This would assign the value $4000 with a memory space attribute of **X** to the symbol A_D_PORT.

SET

EXITM

Syntax:

EXITM

Description:

Exit Macro. The **EXITM** directive will cause immediate termination of a macro expansion. It is useful when used with the conditional assembly directive **IF** to terminate macro expansion when error conditions are detected.

A label is not allowed with this directive.

Examples:

```
CALC  MACRO  XVAL,YVAL
      IF     XVAL<0
      FAIL   'Macro parameter value out of range'
      EXITM  ;Exit macro
      ENDIF
      .
      .
      .
      ENDM
```

DUP, **DUPA**, **DUPC**, **MACRO**

EXTERN

Syntax:

EXTERN [**(***attrib*[**,***attrib*]...**)**] [*mem***:**]*symbol*[**,**[*mem***:**]*symbol*]...

Description:

External Symbol Declaration. The **EXTERN** directive is used to specify that the list of symbols is referenced in the current module, but is not defined within the current module. These symbols must either have been defined outside of any module or declared as globally accessible within another module using the **GLOBAL** directive.

The optional argument *attrib* can be one of the following symbol attributes:

FAR	symbol is long addressable (default)
NEAR	symbol is short addressable
INTERN	use internal busses (X, Y and program address bus) to access extern memory (default)
EXTERN	use external busses to access extern memory. intern and extern only affect the cycle count

mem corresponds to one of the DSP memory spaces (**X**, **Y**, **L**, **P**, **E**).

If the **EXTERN** directive is not used to specify that a symbol is defined externally and the symbol is not defined within the current module, a warning is generated, and an **EXTERN** symbol is inserted.

A label is not allowed with this directive.

Examples:

```
SECTION   FILER
EXTERN    AA,CC,DD        ;defined elsewhere
EXTERN    (near) Y:EE     ;short addressable external
                          ;symbol in Y memory
  .
  .
  .
ENDSEC
```

GLOBAL, **SECTION**

FAIL

Syntax:

FAIL [{*str* | *exp*}[,{*str* | *exp*}]...]

Description:

Programmer Generated Error. The **FAIL** directive will cause an error message to be output by the assembler. The total error count will be incremented as with any other error. The **FAIL** directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the error has been printed. An arbitrary number or strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the generated error.

A label is not allowed with this directive.

Examples:

```
FAIL  'Parameter out of range'
```

MSG, **WARN**

FORCE

Syntax:

FORCE {**NEAR** | **FAR** | **NONE**}

Description:

Set Operand Forcing Mode. The **FORCE** directive causes the assembler to force all immediate, memory, and address operands to the specified mode as if an explicit forcing operator were used. Note that if a relocatable operand value forced short is determined to be too large for the instruction word, an error will occur at link time, not during assembly. Explicit forcing operators override the effect of this directive.

For compatibility reasons the **FORCE** directive also accepts the arguments **SHORT** instead of **NEAR** and **LONG** instead of **FAR**.

A label is not allowed with this directive.

Examples:

```
FORCE  NEAR  ;force operands short
```

<, **>**, **#<**, **#>**

GLOBAL

Syntax:

GLOBAL *symbol*[**,***symbol*]...

Description:

Global Section Symbol Declaration. The **GLOBAL** directive is used to specify that the list of symbols is defined within the current section or module, and that those definitions should be accessible by all sections. This directive is not only valid if used within a program block bounded by the **SECTION** and **ENDSEC** directives, but also valid within the module body. If the symbols that appear in the operand field are not defined in the section, an error will be generated. Symbols that are defined ”global” are accessible from other modules using the **EXTERN** directive.

A label is not allowed with this directive.

Examples:

```
SECTION  IO
GLOBAL   LOOPA     ;LOOPA will be globally
.                  ;accessible by other sections
.
.
ENDSEC
```

EXTERN, **LOCAL**, **SECTION**

GSET

Syntax:

label **GSET** *expression*

GSET *label* *expression*

Description:

Set Global Symbol to a Value. The **GSET** directive is used to assign the value of the expression in the operand field to the label. The **GSET** directive functions somewhat like the **EQU** directive. However, labels defined via the **GSET** directive can have their values redefined in another part of the program (but only through the use of another **GSET** or **SET** directive). The **GSET** directive is useful for resetting a global **GSET** symbol within a scope, where the **SET** symbol would otherwise be considered local. The expression in the operand field of a **GSET** may have forward references.

Examples:

```
COUNT  GSET  0     ; Initialize COUNT
```

EQU, **SET**

IF

Syntax:

IF *expression*
…
[**ELSE**] (the **ELSE** directive is optional)
…
ENDIF

Description:

Conditional Assembly. Part of a program that is to be conditionally assembled must be bounded by an **IF–ENDIF** directive pair. If the optional **ELSE** directive is not present, then the source statements following the **IF** directive and up to the next **ENDIF** directive will be included as part of the source file being assembled only if the *expression* has a nonzero result. If the *expression* has a value of zero, the source file will be assembled as if those statements between the **IF** and the **ENDIF** directives were never encountered. If the **ELSE** directive is present and *expression* has a nonzero result, then the statements between the **IF** and **ELSE** directives will be assembled, and the statements between the **ELSE** and **ENDIF** directives will be skipped. Alternatively, if *expression* has a value of zero, then the statements between the **IF** and **ELSE** directives will be skipped, and the statements between the **ELSE** and **ENDIF** directives will be assembled.

The *expression* must have an absolute integer result and is considered true if it has a nonzero result. The *expression* is false only if it has a result of 0. Because of the nature of the directive, *expression* must be known on pass one (no forward references allowed). **IF** directives can be nested to any level. The **ELSE** directive will always refer to the nearest previous **IF** directive as will the **ENDIF** directive.

A label is not allowed with this directive.

Examples:

```
IF     @LST>0
DUP    @LST          ; Unwind LIST directive stack
NOLIST
ENDM
ENDIF
```

ENDIF

INCLUDE

Syntax:

INCLUDE *string* | **<***string***>**

Description:

Include Secondary File. This directive is inserted into the source program at any point where a secondary file is to be included in the source input stream. The string specifies the filename of the secondary file. The filename must be compatible with the operating system and can include a directory specification.

The file is searched for first in the current directory, unless the **<***string***>** syntax is used, or in the directory specified in *string*. If the file is not found, and the **–I** option was used on the command line that invoked the assembler, then the string specified with the **–I** option is prefixed to *string* and that directory is searched. If the **<***string***>** syntax is given, the file is searched for only in the directories specified with the **–I** option.

A label is not allowed with this directive.

Examples:

```
INCLUDE ’headers/io.asm’     ; Unix example
INCLUDE ’storage\mem.asm’    ; PC example
INCLUDE <data.asm>           ; Do not look in
                             ; current directory
```

LIST

Syntax:

LIST

Description:

List the Assembly. Print the listing from this point on. The **LIST** directive will not be printed, but the subsequent source lines will be output to the source listing. When the **–l** command line option has not been given, the **LIST** directive has no effect.

The **LIST** directive actually increments a counter that is checked for a positive value and is symmetrical with respect to the **NOLIST** directive. Note the following sequence:

```
; Counter value currently 1
LIST                ; Counter value = 2
LIST                ; Counter value = 3
NOLIST              ; Counter value = 2
NOLIST              ; Counter value = 1
```

The listing still would not be disabled until another **NOLIST** directive was issued.

A label is not allowed with this directive.

Examples:

```
IF  LISTON
LIST                   ; Turn the listing back on
ENDIF
```

NOLIST

LOCAL

Syntax:

LOCAL *symbol*[**,***symbol*]...

Description:

Local Section Symbol Declaration. The **LOCAL** directive is used to specify that the list of symbols is defined within the current section, and that those definitions are explicitly local to that section or module. It is useful in cases where a symbol may not be exported outside of the module (as labels on module level are defined ”global” by default). This directive is not only valid if used within a program block bounded by the **SECTION** and **ENDSEC** directives, but also valid within the module body. The **LOCAL** directive must appear in the same scope as where *symbol* is defined in the section. If the symbols that appear in the operand field are not defined in the section, an error will be generated.

A label is not allowed with this directive.

Examples:

```
SECTION  IO
LOCAL  LOOPA        ;LOOPA local to this section
.
.
ENDSEC
```

SECTION, **GLOBAL**

MACRO

Syntax:

name **MACRO** [*dummy_argument_list*]
.
macro_definition_statements
.
.
ENDM

Description:

Macro Definition. The dummy argument list has the form:

[*dumarg*[**,***dumarg*]...]

The required name is the symbol by which the macro will be called.

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the **MACRO** directive, its name, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the **ENDM** directive.

The dummy arguments are symbolic names that the macro processor will replace with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as symbol names. Dummy argument names that are preceded by an underscore are not allowed. In the dummy argument list, the dummy arguments are separated by commas. The dummy argument list is separated from the MACRO directive by one or more blanks.

Macro definitions may be nested but the nested macro will not be defined until the primary macro is expanded.

Chapter 6, *Macro Operations*, contains a complete description of macros.

Examples:

```
SWAP_SYM  MACRO  REG1,REG2       ;swap REG1,REG2
    MOVE  R\?REG1,X0             ;using X0 as temp
    MOVE  R\?REG2,R\?REG1
    MOVE  X0,R\?REG2
    ENDM
```

DUP, **DUPA**, **DUPC**, **DUPF**, **ENDM**

MSG

Syntax:

MSG [{*str* | *exp*}[,{*str* | *exp*}]...]

Description:

Programmer Generated Message. The **MSG** directive will cause a message to be output by the assembler. The error and warning counts will not be affected. The **MSG** directive is normally used in conjunction with conditional assembly directives for informational purposes. The assembly proceeds normally after the message has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the message.

A label is not allowed with this directive.

Examples:

```
MSG  'Generating sine tables'
```

FAIL, **WARN**

NOLIST

Syntax:

NOLIST

Description:

Stop Assembly Listing. Do not print the listing from this point on (including the **NOLIST** directive). Subsequent source lines will not be printed.

The **NOLIST** directive actually decrements a counter that is checked for a positive value and is symmetrical with respect to the **LIST** directive. Note the following sequence:

```
; Counter value currently 1
LIST                ; Counter value = 2
LIST                ; Counter value = 3
NOLIST              ; Counter value = 2
NOLIST              ; Counter value = 1
```

The listing still would not be disabled until another **NOLIST** directive was issued.

A label is not allowed with this directive.

Examples:

```
IF   LISTOFF
NOLIST                    ; Turn the listing off
ENDIF
```

LIST, **OPT**

OPT

Syntax:

OPT *option*[,*option*]... [*comment*]

Description:

Assembler Options. The **OPT** directive is used to designate the assembler options. Assembler options are given in the operand field of the source input file and are separated by commas. For most options there is an equivalent command line option (see Chapter 2). All options have a default condition. Some options are reset to their default condition at the end of pass one. All options can have the prefix **NO** attached to them, which then reverses their meaning.

Options are read left–to–right, i.e. the rightmost takes precedence.

Options can be grouped by function into five different types:

1. Listing options
2. Parsing options
3. Code generation
4. Restriction handling
5. Optimizations

(module)	means: last OPT directive counts
(flow)	means: switchable on the fly, during parsing.

A label is not allowed with this directive.

Listing Options (module)

These options control what is reported in the listing file:

CC	–	Show cycle count
CPP	–	List cpp line information
CTRL	–	List controls
EMPTY	–	List empty lines
EQU	–	List equates

HLL	–	List high level language debug information
MD	–	List macro/dup definitions
MEX	–	List macro/dup expansions
MU	–	List section summary
SDEF	–	List symbol definitions
SECT	–	List section directives
WRAP	–	List wrapped part of source line

Parsing Options (flow)

These options control the parsing of the assembler:

AE	–	Check address expressions (W123)
IC	–	Ignore case in symbol names (before any symbol definition)
MSW	–	Warn on memory space incompatibilities (W124)
SVO	–	Preserve object file on errors (module)
UR	–	Flag unresolved references (W118)
W	–	Display warning messages
W*num*	–	Display warning message number *num*

Code Generation (module)

These options deal with code generation:

CACHE128	–	Select code page size of 128 (**as563** only)
CACHE256	–	Select code page size of 256 (**as563** only)
JMPABS	–	Select the absolute branch mode. (flow) (**as563** only)
JMPREL	–	Select the relative branch mode. (flow) (**as563** only)
PS	–	Pack strings
SBM	–	Create constant values for 16–bit mode (**as563** only)
XLL	–	Generate assembly level debug information

Restriction Handling (module)

These options deal with pipeline restrictions:

RP	–	Remove all pipeline restrictions
RPDO	–	Remove DO/ENDDO pipeline restrictions
RPRN	–	Remove Rn, Nn and Mn pipeline restrictions
RPSP	–	Remove stack restrictions

Optimizations (module)

These options deal with assembler optimizations:

OP	–	Perform all optimizations
OPHLL	–	Move symbolic debug locations
OPJMP	–	Perform branch optimization
OPNOP	–	Remove NOP instructions (flow)
OPPM	–	Move parallelization
OPREP	–	Single instruction DO to REP (flow)
OPSP	–	Split parallel instruction before optimization
OPSPEED	–	Optimize for speed at the cost of code size
ORDER	–	Keep instructions in the same order (flow)

Detailed description of assembler options

Following are descriptions of the individual options. An option is specified **default** if it is the behaviour of the assembler when you specify neither the OPT option nor its corresponding command line option.

AE Check address expressions for appropriate arithmetic operations. For example, this will check that only valid add or subtract operations are performed on address terms. This option is equivalent to W123.

CACHE128 Select code page size of 128 (**as563** only).

CACHE256 Select code page size of 256 (**as563** only).

CC Show cycle count (**–LY**).

CPP List source lines containing C preprocessor line information (#line directives) (**–LI**).

CTRL	List source lines containing assembler controls (**–LC**, default).
EMPTY	List empty source lines (**–LN**).
EQU	List source lines containing assembler equates (lines with EQU or '=') (**–LQ**, default).
HLL	List high level language symbolic debug information (lines with SYMB directive) (**–LS**).
IC	Ignore case in symbol, section, and macro names. This directive must be issued before any symbols, sections, or macros are defined. (same as **–c**)
JMPABS	Select the absolute branch mode (**as563** only) (**–Ja**).
JMPREL	Select the relative branch mode **as563** only) (**–Jr**).
MD	List macro/dup definitions (**–LM**, default).
MEX	List macro/dup expansions (**–LX**, default).
MSW	Issue warning on memory space incompatibilities. This option is equivalent to W124.
MU	List section summary. The size of each section is listed (**–t**).
NOAE	Do not check address expressions. This option is equivalent to NOW123 (**–w123**).
NOCC	Hide cycle count (**–Ly**).
NOCPP	Do not list source lines containing C preprocessor line information (**–Li**, default).
NOCTRL	Do not list source lines containing assembler controls (**–Lc**).
NOEMPTY	Do not list empty source lines (**–Ln**, default).
NOEQU	Do not list source lines containing assembler equates (**–Lq**).
NOHLL	Do not list high level language symbolic debug information (**–Ls**, default).
NOIC	(default) Operate case sensitive in symbol, section, and macro names. This directive must be issued before any symbols, sections, or macros are defined.

NOJMPABS	(default) Default branch mode (**as563** only).
NOJMPREL	(default) Default branch mode (**as563** only).
NOMD	Do not list macro/dup definitions (**–Lm**).
NOMEX	Do not list macro/dup expansions (**–Lx**).
NOMSW	Do not issue warning on memory space incompatibilities. This option is equivalent to NOW124 (**–w124**).
NOMU	(default) Do not list section size summary.
NOOP	Do not perform any optimizations (**–OGJMNPRS**). This is the same as specifying NOOPHLL, NOOPJMP, NOOPNOP, NOOPPM, NOOPSPEED, NOOPREP and NOOPSP.
NOOPHLL	Retain symbolic debug locations (**–OG**, default).
NOOPJMP	Do not perform branch optimization (**–OJ**, default).
NOOPNOP	Do not perform NOP removal (**–ON**, default).
NOOPPM	Do not perform parallel move optimization (**–OM**, default).
NOOPREP	Do not perform single instruction DO to REP optimization (**–OR**, default).
NOOPSP	Do not split parallel move instructions before performing optimizations (**–OS**, default).
NOOPSPEED	Do not perform speed optimization (**–OP**).
NOORDER	End an instruction sequence started by ORDER.
NOPS	Do not pack strings in **DC** and **DCB** directive. Individual bytes in strings will be stored one byte per word.
NORP	Do not generate instructions to accommodate pipeline delay (**–RDRS**, default).
NORPDO	Do not remove DO/ENDDO restrictions (**–RD**, default).
NORPRN	Do not remove Rn, Nn, and Mn register restrictions (**–RR**, default).
NORPSP	Do not remove stack restrictions (**–RS**, default).

NOSDEF	Do not list source lines containing symbol definition directives (**–Le**).
NOSECT	Do not list source lines containing section directives (**–Ld**).
NOSVO	Remove object file on errors. Normally any object file produced by the assembler is preserved if errors occur during assembly. This option must be given before any code or data is generated (**–e**).
NOUR	Do not flag unresolved external references. This option is equivalent to NOW118 (**–w118**).
NOW	Do not print warning messages (**–w**).
NOW*num*	Do not print warning message with number *num* (**–w***num*).
NOWRAP	Do not list wrapped part of source lines (**–Lw**).
NOXLL	(default) Do not generate assembly level debug information.
OP	Perform all optimizations (**–O**). This is the same as specifying OPHLL, OPJMP, OPNOP, OPPM, OPSPEED, OPREP and NOOPSP.
OPHLL	Move symbolic debug locations to perform better parallel move optimization (**–Og**).
OPJMP	Perform branch optimization (**–Oj**). The assembler tries to replace branches with shorter or faster functionally equivalent branches.
OPNOP	Remove existing NOP instructions (**–On**).
OPPM	Perform parallel move optimization (**–Om**).
OPREP	Replace a single instruction DO loop with a REP instruction (**–Or**).
OPSP	Split parallel move instructions before performing optimizations (**–Os**).
OPSPEED	Optimize for speed at the cost of code size (**–Op**, default).
ORDER	Mark the begin of an instruction sequence in which the instructions must be kept in the same order. The instruction sequence must end with an OPT NOORDER directive.

PS	(default) Pack strings in **DC** and **DCB** directive. Individual bytes in strings will be packed into consecutive target words for the length of the string.
RP	Remove all pipeline restrictions. Generate NOP instructions to accommodate pipeline delay. The assembler will output a NOP instruction into the output stream and the issues a warning that it has done so (**–R**).
RPDO	Remove DO/ENDDO restrictions (**–Rd**).
RPRN	Remove Rn, Nn, and Mn register restrictions. If an address register is loaded in one instruction then the contents of the register is not available for use as a pointer until after the next instruction. Ordinarily when the assembler detects this condition it issues a warning message. The **RPRN** option will cause the assembler to output a NOP instruction into the output stream and the assembler issues a warning that it has done so (**–Rr**).
RPSP	Remove stack restrictions (**–Rs**).
SDEF	List source lines containing symbol definition directives (**–LE**, default).
SECT	List source lines containing section directives (**–LD**, default).
SVO	(default) Preserve object file on errors. This option must be given before any code or data is generated.
UR	Generate a warning at assembly time for each unresolved external reference. This option is equivalent to W118.
W	Print all warning messages.
W*num*	Print warning message with number *num*.
WRAP	List wrapped part of source line (**–LW**, default).
XLL	Generate assembly level debug information (**–g**).

Examples:

```
OPT  OPNOP,XLL   ;NOP removal and assembly debug
OPT  RP,MU       ;Remove restrictions,
                 ;list section summary
```

ORG

Syntax:

ORG *mem*[*mapcnt*]**:**[*abs–loc*][**,***overlay*]

ORG *mem*[[**,***name*][**,***attrib*]...]**:**[*abs–loc*][**,***overlay*]

where,
mapcnt is defined as:
[*rlc*][*map*]
or: [*map*][**(***rce***)**]

Description:

Initialize Memory Space and Location Counters. The **ORG** directive is used to determine in which section the code following the directive, up to the next **ORG** directive, will be located. The absolute location of the section in memory can be set using this directive.

To uniquely identify a section between different object files sections can be given names. This name can be any character string, as long as it does not start with a space. When a section does not have a name, it is called a nameless section.

The **ORG** directive has two possibilities. The first is Motorola compatible, the second introduces named sections. The Motorola compatible variant defines a nameless section. The section attributes are as defined by the mapping attribute. There is no possibility to state that this entire section is located in short addressable memory, other than to give an absolute location in short memory. As these sections do not have a name, the linker cannot concatenate these sections with equivalent section definitions between different object files.

The second variant defines a named section. The name gives the linker the possibility to concatenate these sections from different object files. Every definition of a named section should use the same section attributes.

A label is not allowed with this directive.

mem	This defines in which memory space (**X**, **Y**, **L**, **P** or **E**) the section is located. Currently the **E** memory space is not supported.

rlc	Which run–time counter **H**, **L**. or default (if neither H or L is specified), that is associated with *mem*, will be used as the run–time location counter: 1. not giving a location counter equals to using the location counter ’0’, 2. the ’L’ location counter equals to the location counter ’1’, and 3. the ’H’ location counter equals to the location counter ’2’. By using the location counter different sections in the same memory space can be defined, without making them absolute by specifying an absolute location address.
map	Indicates the run–time physical mapping to DSP memory: **I** – internal, **E** – external, **R** – ROM, **A** – port A, **B** – port B. If not present, no explicit mapping is done.
rce	Non–negative absolute integer expression representing the counter number to be used as the run–time location counter. Must be enclosed in parentheses. Should not exceed the value 65535.
name	The name defines the section name. This is a string of printable characters. The only exception is that it may not start with a space. In the latter case the name will not be written into the object file.
attrib	The attributes conform to the possibilities given by the Motorola type ORG, but are written with entire words instead of letters. Possibilities are:

FAR	long addressable
NEAR	short addressable
INTERNAL	internal memory, same as mapping ’I’
EXTERNAL	external memory, same as mapping ’E’
OVERLAY	section is an overlay, either data or code
ABSOLUTE	obsolete, the absolute–location must be an absolute expression
BSS	clear section during startup
CONST	initialize during download, do not generate copy table entry

INIT	initialize section during startup (this attribute is required for P data sections
MAX	common, overlay with other parts with the same name, is implicit a type of ’scratch’ and ’overlay’
SCRATCH	not filled, not cleared on startup. Section can only contain ’ds’, no ’dc’ or the like

The OVERLAY attribute is used by the C compiler to generate overlaid data sections in the static model. It can also be used to generate code overlays. For a code overlay, the attributes BSS, CONST, INIT, MAX and SCRATCH are not allowed and the section must be named and must have an absolute address. Furthermore, another section with the same size must be created where the code will be stored at load time. The name of this section must be the same but with ”_copy” appended. Multiple sections with different names can be created this way and with run–time copying they can be used to create a code overlay. This can be advantageous for time–critical routines that can be exchanged in internal program memory. A small example of code overlaying can be found in the `examples` directory.

abs–loc Initial value to assign to the run–time counter used as the *rlc*. *abs–loc* must be an absolute expression.

overlay The overlay part is not supported. When encountered it is skipped and a warning is issued. Overlay descriptions can be described to the locator using the locator description file.

A previously defined section can be continued by giving its name or, for the Motorola ORG–variant, by giving its memory space and location counter.

A named section cannot be continued by only giving its memory space. This will continue the named section with location counter zero!

Examples:

ORG `P:$1000`

Sets the run–time memory space to P. Selects the default run–time counter (counter 0) associated with P space to use as the run–time location counter and initializes it to $1000.

ORG PHE:

Sets the run–time memory space to P. Selects the H load counter (counter 2) associated with P space to use as the run–time location counter. The H counter will not be initialized, and its last value will be used. Code generated hereafter will be mapped to external (E) memory.

Definitions	**Continuation**
`org  p,”.text”,near:$0`	`org  p,”.text”:`
`org  x,”.data”,external:`	`org  x,”.data”:`
`org  pla:$300`	`org  pl:`
`org  xa(10):`	`org  x(10):`
`org  lb(1):`	`org  ll:`
`org  y(0):`	`org  y:`

Chapter *Software Concept* and Appendix *Migration from Motorola CLAS*.

PAGE

Syntax:

PAGE [*exp1*[,*exp2*,...,*exp5*]]

Description:

Top of Page/Size Page. The **PAGE** directive has two forms:

1. If no arguments are supplied, then the assembler will advance the listing to the top of the next page. In this case, the **PAGE** directive will not be output.
2. The **PAGE** directive with arguments can be used to specify the printed format of the output listing. Arguments may be any positive absolute integer expression. The arguments in the operand field (as explained below) are separated by commas. Any argument can be left as the default or last set value by omitting the argument and using two adjacent commas. The **PAGE** directive with arguments will not cause a page eject and will be printed in the source listing.

A label is not allowed with this directive.

The arguments in order are:

PAGE_WIDTH *exp1*

Page width in terms of number of output columns per line (default 80, min 1, max 255).

PAGE_LENGTH *exp2*

Page length in terms of total number of lines per page (default 66, min 10, max 255). As a special case a page length of 0 (zero) turns off all headers, titles, subtitles, and page breaks.

BLANK_TOP *exp3*

Blank lines at top of page. (default 0, min 0, max see below).

BLANK_BOTTOM *exp4*

Blank lines at bottom of page. (default 0, min 0, max see below).

BLANK_LEFT *exp5*

Blank left margin. Number of blank columns at the left of the page. (default 0, min 0, max see below).

The following relationship must be maintained:

BLANK_TOP + BLANK_BOTTOM <= PAGE_LENGTH – 10

BLANK_LEFT < PAGE_WIDTH

Examples:

```
PAGE  132,,3,3   ;Set width to 132,
                 ;3 line top/bottom margins
PAGE             ;Page eject
```

PMACRO

Syntax:

PMACRO *symbol*[,*symbol*]...

Description:

Purge Macro Definition. The specified macro definition will be purged from the macro table, allowing the macro table space to be reclaimed.

A label is not allowed with this directive.

Examples:

```
PMACRO   MAC1,MAC2
```

This statement would cause the macros named MAC1 and MAC2 to be purged.

MACRO

PRCTL

Syntax:

PRCTL *exp*|*string*[,*exp*|*string*]...

Description:

Send Control String to Printer. **PRCTL** simply concatenates its arguments and ships them to the listing file (the directive line itself is not printed unless there is an error). *exp* is a byte expression and *string* is an assembler string. A byte expression would be used to encode non–printing control characters, such as ESC. The string may be of arbitrary length, up to the maximum assembler–defined limits.

PRCTL may appear anywhere in the source file and the control string will be output at the corresponding place in the listing file. However, if a **PRCTL** directive is the last line in the last input file to be processed, the assembler insures that all error summaries, symbol tables, and cross–references have been printed before sending out the control string. This is so a **PRCTL** directive can be used to restore a printer to a previous mode after printing is done. Similarly, if the **PRCTL** directive appears as the first line in the first input file, the control string will be output before page headings or titles.

The **PRCTL** directive only works if the **–l** command line option is given; otherwise it is ignored.

A label is not allowed with this directive.

Examples:

```
PRCTL  $1B,'E'  ;Reset HP LaserJet printer
```

RADIX

Syntax:

RADIX *expression*

Description:

Change Input Radix for Constants. Changes the input base of constants to the result of *expression*. The absolute integer expression must evaluate to one of the legal constant bases (2, 10, or 16). The default radix is 10. The **RADIX** directive allows the programmer to specify constants in a preferred radix without a leading radix indicator. The radix prefix for base 10 numbers is the grave accent (‘). Note that if a constant us used to alter the radix, it must be in the appropriate input base at the time the **RADIX** directive is encountered.

A label is not allowed with this directive.

Examples:

```
_RAD10      DC  10   ; Evaluates to hex A
    RADIX   2
_RAD2       DC  10   ; Evaluates to hex 2
    RADIX   ‘16
_RAD16      DC  10   ; Evaluates to hex 10
    RADIX   3        ; Bad radix expression
```

SCSJMP

Syntax:

SCSJMP {NEAR | FAR | NONE}

Description:

Set Structured Control Statement Branching Mode. The **SCSJMP** directive analogous to the **FORCE** directive, but it only applies to branches generated automatically by structured control statements (see Chapter 8). There is no explicit way, as with a forcing operator, to force a branch short or long when it is produced by a structured control statement. This directive will cause all branches resulting from subsequent structured control statements to be forced to the specified mode.

Just like the **FORCE** pseudo–op, errors can result if a value is too large to be forced short. For relocatable code, the error may not occur until the linking phase.

For compatibility reasons the **SCSJMP** directive also accepts the arguments **SHORT** instead of **NEAR** and **LONG** instead of **FAR**.

A label is not allowed with this directive.

Examples:

```
SCSJMP   NEAR   ;force all subsequent SCS
                ;jumps short
```

FORCE, **SCSREG**

SCSREG

Syntax:

SCSREG [*srcreg*[,*dstreg*[,*tmpreg*[,*extreg*]]]]

Description:

Reassign Structured Control Statement Registers. The **SCSREG** directive reassigns the registers used by structured control statement (SCS) directives (see Chapter 8). It is convenient for reclaiming default SCS registers when they are needed as application operands within a structured control construct. *srcreg* is ordinarily the source register for SCS data moves. *dstreg* is the destination register. *tmpreg* is a temporary register for swapping SCS operands. *extreg* is an extra register for complex SCS operations. With no arguments **SCSREG** resets the SCS registers to their default assignments.

The **SCSREG** directive should be used judiciously to avoid register context errors during SCS expansion. Source and destination registers may not necessarily be used strictly as source and destination operands. The assembler does no checking of reassigned registers beyond validity for the target processor. Errors can result when a structured control statement is expanded and an improper register reassignment has occurred. It is recommended that the **MEX** option (see the **OPT** directive) be used to examine structured control statement expansion for relevant constructs to determine default register usage and applicable reassignment strategies.

A label is not allowed with this directive.

Examples:

```
SCSREG   Y0,B     ;Reassign SCS source and
                  ;dest. registers
```

OPT (MEX), **SCSJMP**

SECTION

Syntax:

SECTION *scope_name*
.
.
section source statements
.
.
ENDSEC

Description:

Start Scope. The **SECTION** directive defines the start of a scope. All symbols that are defined within a scope have the *scope_name* associated with them as their scope name. This serves to protect them from like–named symbols elsewhere in the program. A symbol defined inside any given scope is private to that scope.

Symbols within a scope are generally distinct from other symbols used elsewhere in the source program, even if the symbol name is the same. This is true as long as the scope name associated with each symbol is unique and the symbol is not declared public (**GLOBAL**). Symbols that are defined outside of a scope are considered global symbols and have no explicit scope name associated with them. Global symbols may be referenced freely from inside or outside of any scope, as long as the global symbol name does not conflict with another symbol by the same name in a given scope.

The division of a program into scopes controls not only labels and symbols, but also macros and **DEFINE** directive symbols. Macros defined within a scope are private to that scope and are distinct from macros defined in other scopes even if they have the same macro name. Macros defined outside of scopes are considered global and may be used within any scope. Similarly, **DEFINE** directive symbols defined within a scope are private to that scope and **DEFINE** directive symbols defined outside of any scope are globally applied. There are no directives that correspond to **GLOBAL** for macros or **DEFINE** symbols, and therefore, macros and **DEFINE** symbols defined in a scope can never be accessed globally. If global accessibility is desired, the macros and **DEFINE** symbols should be defined outside of any scope.

Section scopes can be nested to any level. When the assembler encounters a nested scope, the current scope is stacked and the new scope is used. When the **ENDSEC** directive of the nested scope is encountered, the assembler restores the old scope and uses it. The **ENDSEC** directive always applies to the most previous **SECTION** directive. Nesting scopes provides a measure of scoping for symbol names, in that symbols defined within a given scope are visible to other scopes nested within it. For example, if scope B is nested inside scope A, then a symbol defined in scope A can be used in scope B without **GLOBAL**ing in scope A or **EXTERN**ing in scope B.

Scopes may also be split into separate parts. That is, *scope_name* can be used multiple times with **SECTION** and **ENDSEC** directive pairs. If this occurs, then these separate (but identically named) scopes can access each others symbols freely without the use of the **GLOBAL** and **EXTERN** directives. If the **GLOBAL** and **EXTERN** directives are used within one scope, they apply to all scopes with the same scope name. The reuse of the scope name is allowed to permit the program source to be arranged in an arbitrary manner (for example, all statements that reserve X space storage locations grouped together), but retain the privacy of the symbols for each scope.

A label is not allowed with this directive.

Examples:

```
SECTION  TABLES      ;TABLES will be the scope name
```

ORG, **GLOBAL**, **LOCAL**, **EXTERN**

SET

Syntax:

label **SET** *expression*

SET *label* *expression*

Description:

Set Symbol to a Value. The **SET** directive is used to assign the value of the expression in the operand field to the label. The **SET** directive functions somewhat like the **EQU** directive. However, labels defined via the **SET** directive can have their values redefined in another part of the program (but only through the use of another **SET** directive). The **SET** directive is useful in establishing temporary or reusable counters within macros. The expression in the operand field of a **SET** may have forward references.

Examples:

```
COUNT  SET  0    ; Initialize COUNT
```

EQU, **GSET**

STITLE

Syntax:

STITLE [*string*]

Description:

Initialize Program Sub–Title. The **STITLE** directive initializes the program subtitle to the *string* in the operand field. The subtitle will be printed on the top of all succeeding pages until another **STITLE** directive is encountered. The subtitle is initially blank. The **STITLE** directive will not be printed in the source listing. An **STITLE** directive with no string argument causes the current subtitle to be blank.

If the page width is too small for the title to fit in the header, it will be truncated.

A label is not allowed with this directive.

Examples:

```
STITLE  'COLLECT SAMPLES'
```

TITLE

SYMB

Syntax:

SYMB *string*, *expression* [, *abs_expr*] [, *abs_expr*]

Description:

The **SYMB** directive is used for passing high–level language symbolic debug information via the assembler (and linker/locator) to the debugger. *expression* can be any expression. *abs_expr* can be any expression resulting in an absolute value.

The SYMB directive is not meant for 'hand coded' assembly files. It is documented for completeness only and is supposed to be 'internal' to the tool chain.

TABS

Syntax:

TABS *tabstops*

Description:

Set Listing Tab Stops. The **TABS** directive allows resetting the listing file tab stops from the default value of 8.

A label is not allowed with this directive.

Examples:

```
TABS   4    ;Set listing file tab stops to 4
```

TITLE

Syntax:

TITLE [*string*]

Description:

Initialize Program Title. The **TITLE** directive initializes the program title to the *string* in the operand field. The program title will be printed on the first page of the list file. The title is initially blank. The **TITLE** directive will not be printed in the source listing. A **TITLE** directive with no string argument causes the current title to be blank. For titles on succeeding pages use the **STITLE** directive.

If the page width is too small for the title to fit in the header, it will be truncated.

A label is not allowed with this directive.

Examples:

```
TITLE  'FIR FILTER'
```

STITLE

UNDEF

Syntax:

UNDEF *symbol*

Description:

Undefine DEFINE Symbol. The **UNDEF** directive causes the substitution string associated with *symbol* to be released, and *symbol* will no longer represent a valid **DEFINE** substitution. See the **DEFINE** directive for more information.

A label is not allowed with this directive.

Examples:

```
UNDEF  DEBUG   ;Undefines the DEBUG substitution
               ;string
```

DEFINE

VOID

Syntax:

label **VOID** *register* [, *register*]...

Description:

The **VOID** directive defines which registers may contain unknown values when the *label* is reached. The assembler may use this information when optimizing the assembly program.

Examples:

When you use the **VOID** directive at a DO loop label, the assembler tries to duplicate instructions from the loop head to the loop tail and prior to the loop.

The following example,

```
    DO    #10,label
    move  x:(r1)+,a
    ; ... some other code
    lsl   b
label:    void  r1,a
```

can be optimized to:

```
    move  x:(r1)+,a
    DO    #10,label
    ; ... some other code
    lsl   b  x:(r1)+,a
label:    void  r1,a
```

WARN

Syntax:

WARN [{*str* | *exp*}[**,**{*str* | *exp*}]...]

Description:

Programmer Generated Warning. The **WARN** directive will cause a warning message to be output by the assembler. The total warning count will be incremented as with any other warning. The **WARN** directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the warning has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the generated warning.

A label is not allowed with this directive.

Examples:

```
WARN  'parameter too large'
```

FAIL, **MSG**

CHAPTER 8

STRUCTURED CONTROL STATEMENTS

TASKING

CHAPTER 8

8.1 INTRODUCTION

An assembly language provides an instruction set for performing certain rudimentary operations. These operations in turn may be combined into control structures such as loops (FOR, REPEAT, WHILE) or conditional branches (IF–THEN, IF–THEN–ELSE). The assembler, however, accepts formal, high–level directives that specify these control structures, generating the appropriate assembly language instructions for their efficient implementation. This use of structured control statement directives improves the readability of assembly language programs, without compromising the desirable aspects of programming in an assembly language.

8.2 STRUCTURED CONTROL DIRECTIVES

The following directives are used for structured control. Note the leading period, which distinguishes these keywords from other directives and mnemonics. Structured control directives may be specified in either upper or lower case, but they must appear in the opcode field of the instruction line (e.g. they must be preceded either by a label, a space, or a tab).

.BREAK	**.ENDI**	**.LOOP**
.CONTINUE	**.ENDL**	**.REPEAT**
.ELSE	**.ENDW**	**.UNTIL**
.ENDF	**.FOR**	**.WHILE**
	.IF	

In addition, the following keywords are used in structured control statements:

AND	**DOWNTO**	**TO**
BY	**OR**	
DO	**THEN**	

AND, DO, and OR are reserved assembler instruction mnemonics.

8.3 SYNTAX

The formats for the **.BREAK**, **.CONTINUE**, **.FOR**, **.IF**, **.LOOP**, **.REPEAT**, and **.WHILE** statements are given in the following sub–sections. Syntactic variables used in the formats are defined as follows:

expression	A simple or compound expression (section 8.4).
stmtlist	Zero or more assembler directives, structured control statements, or executable instructions.

Note that an assembler directive (Chapter 7) occurring within a structured control statement is examined exactly once –– at assembly time. Thus the presence of a directive within a **.FOR**, **.LOOP**, **.REPEAT**, or **.WHILE** statement does not imply repeated occurrence of an assembler directive; nor does the presence of a directive within an **.IF–THEN–.ELSE** structured control statement imply conditional assembly.

op1	A user–defined operand whose register/memory location holds the **.FOR** loop counter. The effective address must use a memory alterable addressing mode (e.g. it cannot be an immediate value).
op2	The initial value of the **.FOR** loop counter. The effective address may be any mode, and may represent an arbitrary assembler expression (Section 5.2, *Expressions*).
op3	The terminating value of the **.FOR** loop counter. The effective address may be any mode, and may represent an arbitrary assembler expression (Section 5.2, *Expressions*).
op4	The step (increment/decrement) of the **.FOR** loop counter each time through the loop. If not specified, it defaults to a value of #1. The effective address may be any mode, and may represent an arbitrary assembler expression (Section 5.2, *Expressions*).
cnt	The terminating value in a **.LOOP** statement. This can be any arbitrary assembler expression (Section 5.2, *Expressions*).

All structured control statements may be followed by normal assembler comments on the same logical line.

8.3.1 .BREAK STATEMENT

Syntax:

.BREAK

Function:

The **.BREAK** statement causes an immediate exit from the innermost enclosing loop construct (**.WHILE**, **.REPEAT**, **.FOR**, **.LOOP**).

A **.BREAK** statement does not exit an **.IF–THEN–.ELSE** construct. If a **.BREAK** is encountered with no loop statement active, a warning is issued.

.BREAK should be used with care near **.ENDL** directives or near the end of DO loops. It generates a jump instruction which is illegal in those contexts.

Examples:

```
.WHILE    x:(r1)+ <GT> #0;loop until zero is found
     .
     .
     .
.IF  <cs> ;carry set?
.BREAK    ;causes exit from WHILE loop
.ENDI
     .
     .    ;any instructions here are skipped
     .
.ENDW
;execution resumes here after .BREAK
```

8.3.2 .CONTINUE STATEMENT

Syntax:

.CONTINUE

Function:

The **.CONTINUE** statement causes the next iteration of a looping construct (**.WHILE**, **.REPEAT**, **.FOR**, **.LOOP**) to begin. This means that the loop expression or operand comparison is performed immediately, bypassing any subsequent instructions.

If a **.CONTINUE** is encountered with no loop statement active, a warning is issued.

.CONTINUE should be used with care near **.ENDL** directives or near the end of DO loops. It generates a jump instruction which is illegal in those contexts.

One or more **.CONTINUE** directives inside a **.LOOP** construct will generate a NOP instruction just before the loop address.

Examples:

```
.REPEAT
     .
     .
     .
.IF  <cs> ;carry set?
.CONTINUE ;causes immediate jump to .UNTIL
.ENDI
     .
     .    ;any instructions here are skipped
     .
.UNTIL    x:(r1)+ <EQ> #0;evaluation here after
                         ;.CONTINUE
```

8.3.3 .FOR STATEMENT

Syntax:

.FOR *op1* **=** *op2* {**TO** | **DOWNTO**} *op3* [**BY** *op4*] [**DO**]

stmtlist

.ENDF

Function:

Initialize *op1* to *op2* and perform *stmtlist* until *op1* is greater (**TO**) or less than (**DOWNTO**) *op3*. Makes use of a user–defined operand, *op1*, to serve as a loop counter. **.FOR–TO** allows counting upward, while **.FOR–DOWNTO** allows counting downward. The programmer may specify an increment/decrement step size in *op4*, or select the default step size of #1 by omitting the **BY** clause. A **.FOR–TO** loop is not executed if *op2* is greater than *op3* upon entry to the loop. Similarly, a **.FOR–DOWNTO** loop is not executed if *op2* is less then *op3*.

op1 must be a writable register or memory location. It is initialized at the beginning of the loop, and updated at each pass through the loop. Any immediate operands must be preceded by a pound sign (**#**). Memory references must be preceded by a memory space qualifier (X:, Y:, or P:). L memory references are not allowed. Operands must be or refer to single–word values.

The logic generated by the **.FOR** directive makes use of several DSP data registers (A, X0, Y0, Y1). In fact, two data registers are used to hold the step and target values, respectively, throughout the loop; they are never reloaded by the generated code. It is recommended that these registers not be used within the body of the loop, or that they be saved and restored prior to loop evaluation.

The **DO** keyword is optional.

Examples:

```
.FOR X:CNT = #0 TO Y:(targ*2)+114  ;loop on X:CNT
     .
     .
     .
.ENDF
```

8.3.4 .IF STATEMENT

Syntax:

.IF *expression* [**THEN**]
stmtlist
[**.ELSE**
stmtlist]
.ENDI

Function:

If *expression* is true, execute *stmtlist* following **THEN** (the keyword **THEN** is optional); if *expression* is false, execute *stmtlist* following **.ELSE**, if present; otherwise, advance to the instruction following **.ENDI**.

In the case of nested **.IF–THEN–.ELSE** statements, each **.ELSE** refers to the most recent **.IF–THEN** sequence.

Examples:

```
.IF  <EQ>      ; zero bit set?
     .
     .
     .
.ENDI
```

8.3.5 .LOOP STATEMENT

Syntax:

.LOOP *cnt*
stmtlist
.ENDL

Function:

Execute *stmtlist cnt* times. this is similar to the **.FOR** loop construct, except that the initial counter and step value are implied to be #1. It is actually a shorthand method for setting up a hardware DO loop on the DSP, without having to worry about addressing modes or label placement.

Since the **.LOOP** statement generates instructions for a hardware DO loop, the same restrictions apply as to the use of certain instructions near the end of the loop, nesting restrictions, etc.

One or more **.CONTINUE** directives inside a **.LOOP** construct will generate a NOP instruction just before the loop address.

Examples:

```
.LOOP  LPCNT   ;hardware loop LPCNT times
     .
     .
     .
.ENDL
```

8.3.6 .REPEAT STATEMENT

Syntax:

.REPEAT
stmtlist
.UNTIL *expression*

Function:

stmtlist is executed repeatedly until *expression* is true. When expression becomes true, advance to the next instruction following **.UNTIL**.

The *stmtlist* is executed at least once, even if *expression* is true upon entry to the **.REPEAT** loop.

Examples:

```
.REPEAT
          .
          .
          .
.UNTIL    x:(r1)+ <EQ> #0;loop until zero is found
```

8.3.7 .WHILE STATEMENT

Syntax:

.WHILE *expression* [**DO**]

stmtlist

.ENDW

Function:

The *expression* is tested before execution of *stmtlist*. While *expression* remains true, *stmtlist* is executed repeatedly. When *expression* evaluates false, advance to the instruction following the **.ENDW** statement.

If *expression* is false upon entry to the **.WHILE** loop, *stmtlist* is not executed; execution continues after the **.ENDW** directive.

The **DO** keyword is optional.

Examples:

```
.WHILE    x:(r1)+ <GT> #0   ;loop until zero is found
     .
     .
     .
.ENDW
```

8.4 SIMPLE AND COMPOUND EXPRESSIONS

Expressions are an integral part of **.IF**, **.REPEAT**, and **.WHILE** statements. Structured control statement expressions should not be confused with the assembler expressions discussed in section 5.2, *Expressions*. The latter are evaluated at assembly time and will be referred to here as ”assembler expressions”; they can serve as operands in structured control statement expressions. The structured control statement expressions described below are evaluated at run–time and will be referred to in the following discussion simply as ”expressions”.

A structured control statement expression may be simple or compound. A compound expression consists of two or more simple expressions joined by either **AND** or **OR** (but not both in a single compound expression).

8.4.1 SIMPLE EXPRESSIONS

Simple expressions are concerned with the bits of the Condition Code Register (CCR). These expressions are of two types. The first type merely tests conditions currently specified by the contents of the CCR (section 8.4.1.1). The second type sets up a comparison of two operands to set the condition codes, and afterwards tests the codes (section 8.4.1.2).

8.4.1.1 CONDITION CODE EXPRESSIONS

A variety of tests (identical to those in the Jcc instruction) may be performed, based on the CCR condition codes. The condition codes, in this case, are preset by either a user–generated instruction or a structured operand–comparison expression (section 8.4.1.2). Each test is expressed in the structured control statement by a mnemonic enclosed in angle brackets.

The following condition code mnemonics can be used:

<CC> –	carry clear
<CS> –	carry set
<EC> –	extension clear
<EQ> –	equal
<ES> –	extension set
<GE> –	greater or equal
<GT> –	greater than
<HS> –	higher or same
<LC> –	limit clear
<LE> –	less equal
<LO> –	lower
<LS> –	lmit set
<LT> –	less than
<MI> –	minus
<NE> –	not equal
<NN> –	not normalized
<NR> –	normalized
<PL> –	plus

When processed by the assembler, the expression generates an inverse conditional jump to beyond the matching **.ENDx/.UNTIL** directive. For example:

```
     .IF  <EQ> ;zero bit set?
+    bne  Z_L00002  ;code generated by assembler
     CLR  D1   ;user code
     .ENDI
+    Z_L00002       ;assembler-generated label
     .REPEAT        ;subtract until D0 < D7
+    Z_L00034       ;assembler-generated label
     SUB  D7,D0;user code
     .UNTIL    <LT>
+    bge  Z_L00034  ;code generated by assembler
```

8.4.1.2 OPERAND COMPARISON EXPRESSIONS

Two operands may be compared in a simple expression, with subsequent transfer of control based on that comparison. Such a comparison takes the form:

op1 cc op2

where *cc* is a condition mnemonic enclosed in angle brackets (as described in section 8.4.1.1), and *op1* and *op2* are register or memory references, symbols, or assembler expressions. When processed by the assembler, the operands are arranged such that a compare/jump sequence of the following form always results:

```
CMP          reg1,reg2
(J|B)cc      label
```

where the jump conditional is the inverse of *cc*. Ordinarily *op1* is moved to the *reg1* data register and *op2* is moved to the *reg2* data register prior to the compare. This is not always the case, however: if *op1* happens to be *reg2* and *op2* is *reg1*, an intermediate register is used as a scratch register. In any event, worst case code generation for a given operand comparison expression is generally two moves, a compare, and a conditional jump.

Jumps or branches generated by structured control statements are forced long because the number and address of intervening instructions between a control statement and its termination are not known by the assembler. The programmer may circumvent this behavior by use of the **SCSJMP** directive (see Chapter 7).

Any immediate operands must be preceded by a pound sign (**#**). Memory references must be preceded by a memory space qualifier (X:, Y:, or P:). L memory references are not allowed. Operands must be or refer to single–word values.

Note that values in the *reg1* and *reg2* data registers are not saved before expression evaluation. This means that any user data in those registers will be overwritten each time the expression is evaluated at runtime. The programmer should take care either to save needed contents of the registers, reassign data registers using the **SCSREG** directive, or not use them at all in the body of the particular structured construct being executed. The data registers used by the structured control statements are A, X0, Y0 and Y1.

8.4.2 COMPOUND EXPRESSIONS

A compound expression consists of two or more simple expressions (section 8.4.1) joined by a logical operator (**AND** or **OR**). The boolean value of the compound expression is determined by the boolean values of the simple expressions and the nature of the logical operator. Note that the result of mixing logical operators in a compound expression is undefined:

```
.IF  X1 <GT> B AND <LS> AND R1 <NE> R2  ;this is OK
.IF  X1 <LE> B AND <LC> OR R5 <GT> R6   ;undefined
```

The simple expressions are evaluated left to right. Note that this means the result of one simple expression could have an impact on the result of subsequent simple expression, because of the condition code settings stemming from the assembler–generated compare.

If the compound expression is an **AND** expression and one of the simple expressions is found to be false, any further simple expressions are not evaluated. Likewise, if the compound expression is an **OR** expression and one of the simple expressions is found to be true, any further simple expressions are not evaluated. In these cases, the compound expression is either false or true, respectively, and the condition codes reflects the result of the last simple expression evaluated.

8.5 STATEMENT FORMATTING

The format of structured control statements differs somewhat from normal assembler usage. Whereas a standard assembler line is split into fields separated by blanks or tabs, with no white space inside the fields, structured control statement formats vary depending on the statement being analyzed. In general, all structured control directives are placed in the opcode field (with an optional label in the label field) and white space separates all distinct fields in the statement. Any structured control statement may be followed by a comment on the same logical line.

8.5.1 EXPRESSION FORMATTING

Given an expression of the form:

op1 **<LT>** *op2* **OR** *op3* **<GE>** *op4*

there must be white space (blank, tab) between all operands and their associated operators, including boolean operators in compound expressions. Moreover, there must be white space between the structured control directive and the expression, and between the expression and any optional directive modifier (**THEN**, **DO**). An assembler expression (section 5.2, *Expressions*) used as an operand in a structured control statement expression must not have white space in it, since it is parsed by the standard assembler evaluation routines:

```
.IF  #@CVI@SQT(4.0)) <GT> #2   ;no white space in
                               ;first operand
```

8.5.2 .FOR/.LOOP FORMATTING

The **.FOR** and **.LOOP** directives represent special cases. The **.FOR** structured control statement consists of several fields:

.FOR *op1* **=** *op2* **TO** *op3* **BY** *op4* **DO**

There must be white space between all operands and other syntactic entities such as **=**, **TO**, **BY**, and **DO**. As with expression formatting, an assembler expression used as an operand must not have white space in it:

```
.FOR X:CNT = #0 TO Y:(targ*2)+1 BY
#@CVI@POW(2.0,@CVF(R)))
```

In the example above, the **.FOR** loop operands represented as assembler expressions (symbol, function) do not have embedded white space, whereas the loop operands are always separated from structured control statement keywords by white space.

The count field of a **.LOOP** statement must be separated from the **.LOOP** directive by white space. The count itself may be any arbitrary assembler expression, and therefore must not contain embedded blanks.

8.5.3 ASSEMBLY LISTING FORMAT

Structured control statements begin with the directive in the opcode field; any optional label is output in the label field. The rest of the statement is left as is in the operand field, except for any trailing comment; the X and Y data movement fields are ignored. Comments following the statement are output in the comment field (see Chapter 4).

Statements are expanded using the macro facilities of the assembler. Thus the generated code can be sent to the listing by specifying the **MEX** assembler option, via the **OPT** directive (Chapter 7).

8.6 EFFECTS ON THE PROGRAMMER'S ENVIRONMENT

During assembly, global labels beginning with "**Z_L**" are generated. They are stored in the symbol table and should not be duplicated in user–defined labels. These non–local labels ordinarily are not visible to the programmer, but there can be problems when local (underscore) labels are interspersed among structured control statements. The **SCL** option (see the **OPT** directive, Chapter 7) causes the assembler to maintain the current local label scope when a structured control statement label is encountered.

In the **.FOR** loop, *op1* is a user–defined symbol. When exiting the loop, the memory/register assigned to this symbol contains the value which caused the exit from the loop.

A compare instruction is produced by the assembler whenever two operands are tested in a structured statement. At runtime, these assembler–generated instructions set the condition codes of the CCR (in the case of a loop, the condition codes are set repeatedly). Any user–written code either within or following a structured statement that references CCR directly (move) or indirectly (conditional) jump/transfer) should be attentive to the effect of these instructions.

Jumps or branches generated by structured control statements are forced long because the number and address of intervening instructions between a control statement and its termination are not known by the assembler. The programmer may circumvent this behavior by use of the **SCSJMP** directive (see Chapter 7).

In all structured control statements except those using only a single condition code expression, registers are used to set up the required counters and comparands. In some cases, these registers are effectively reserved; the **.FOR** loop uses two data registers to hold the step and target values, respectively, and performs no save/restore operations on these registers. The assembler, in fact, does no save/restore processing in any structured control operation; it simply moves the operands into appropriate registers to execute the compare. The following registers are used by the assembler in support of structured control statements:

A, X0, Y0, Y1

The **SCSREG** directive (Chapter 7) may be used to reassign structured control statement registers. The **MEX** assembler option (see the **OPT** directive, Chapter 7) may be used to send the assembler–generated code to the listing file for examination of possible register use conflicts.

CHAPTER

INSTRUCTION SET

9

TASKING

CHAPTER 9

9.1 INTRODUCTION

The **as56** DSP5600x assembler accepts all the assembly language instruction mnemonics defined for the DSP5600x. The mnemonics are listed in the tables below.

For a complete list of all DSP5600x instructions with mnemonics, operands, opcode format and states refer to Motorola's *DSP56000 Digital Signal Processor Family* Manual. For the DSP563xx refer to Motorola's *DSP56300 24–Bit Digital Signal Processor Family* Manual. For the DSP566xx refer to Motorola's *DSP56600 Digital Signal Processor Family* Manual. The addressing modes used with the instructions and the meaning and use of the Condition Codes are identical to the corresponding Motorola features.

Some instructions in the tables below are only available in one of the DSP56xxx families. The other instructions are available in both the DSP5600x, DSP563xx and the DSP566xx. However, some instructions may have a different implementation. Refer to the corresponding Digital Signal Processor Family Manual for details.

9.2 THE INSTRUCTION SET

This section contains a summary of the DSP56xxx instruction set. The instruction set can be subdivided in instruction classes as follows.

9.2.1 ARITHMETIC INSTRUCTIONS

Mnemonic	Operation
abs	Absolute Value
adc	Add Long with Carry
add	Add
addl	Shift Left and Add
addr	Shift Right and Add
asl	Arithmetic Shift Left
asr	Arithmetic Shift Right
clr	Clear an Operand
cmp	Compare

Mnemonic	Operation
cmpm	Compare Magnitude
cmpu	Compare Unsigned (DSP563xx, DSP566xx only)
dec	Decrement
div	Divide Iteration
dmacss	Double (Multi) precision oriented MAC (signed x signed) (DSP563xx, DSP566xx only)
dmacsu	Double (Multi) precision oriented MAC (signed x unsigned) (DSP563xx, DSP566xx only)
dmacuu	Double (Multi) precision oriented MAC (unsigned x unsigned) (DSP563xx, DSP566xx only)
inc	Increment by One
mac	Signed Multiply–Accumulate
maci	Signed Multiply–Accumulate (immediate operand) (DSP563xx, DSP566xx only)
macr	Signed Multiply–Accumulate and Round
macri	Signed Multiply–Accumulate and Round (immediate operand) (DSP563xx, DSP566xx only)
macsu	Mixed mode Multiply–Accumulate (signed x unsigned) (DSP563xx, DSP566xx only)
macuu	Mixed mode Multiply–Accumulate (unsigned x unsigned) (DSP563xx, DSP566xx only)
max	Transfer By Signed Value (DSP563xx, DSP566xx only)
maxm	Transfer By Magnitude (DSP563xx, DSP566xx only)
mpy	Signed Multiply
mpyi	Signed Multiply (immediate operand) (DSP563xx, DSP566xx only)
mpyr	Signed Multiply and Round
mpyri	Signed Multiply and Round (immediate operand) (DSP563xx, DSP566xx only)
mpysu	Mixed mode Multiply (DSP563xx, DSP566xx only)
neg	Negate Accumulator
norm	Normalize (DSP563xx, DSP566xx only)
normf	Fast Accumulator Normalize (DSP563xx, DSP566xx only)
rnd	Round
sbc	Subtract Long with Carry

Mnemonic	Operation
sub	Subtract
subl	Shift Left and Subtract
subr	Shift Right and Subtract
t*cc*	Transfer Conditionally
tfr	Transfer Data ALU Register
tst	Test an Operand

Table 9–1: Arithmetic instructions

9.2.2 LOGICAL INSTRUCTIONS

Mnemonic	Operation
and	Logical ANDf
andi	AND Immediate to Control Register
clb	Count Leading Bits (DSP563xx, DSP566xx only)
eor	Logical Exclusive OR
extract	Extract Bit Field (DSP563xx, DSP566xx only)
extractu	Extract Unsigned Bit Field (DSP563xx, DSP566xx only)
insert	Insert Bit Field (DSP563xx, DSP566xx only)
lsl	Logical Shift Left
lsr	Logical Shift Right
merge	Merge Two Half Words (DSP563xx, DSP566xx only)
not	Logical Complement
or	Logical Inclusive OR
ori	OR Immediate to Control Register
rol	Rotate Left
ror	Rotate Right

Table 9–2: Logical instructions

9.2.3 BIT MANIPULATION INSTRUCTIONS

Mnemonic	Operation
bchg	Bit Test and Change
bclr	Bit Test and Clear
bset	Bit Test and Set
btst	Bit Test on Memory and Registers

Table 9–3: Bit manipulation instructions

9.2.4 LOOP INSTRUCTIONS

Mnemonic	Operation
brk*cc*	Conditionally Exit from Hardware Loop (DSP563xx, DSP566xx only)
do	Start Hardware Loop
dor	Start Hardware Loop to PC–Related End–Of–Loop Location (DSP563xx only)
do forever	Start Forever Hardware Loop (DSP563xx, DSP566xx only)
dor forever	Start Forever Hardware Loop to PC–Related End–Of–Loop Location (DSP563xx only)
enddo	Exit from Hardware Loop

Table 9–4: Loop instructions

9.2.5 MOVE INSTRUCTIONS

Mnemonic	Operation
lea	Load Effective Address (DSP563xx only)
lra	Load PC–Relative Address (DSP563xx, DSP566xx only)
lua	Load Updated Address
move	Move Data Register
movec	Move Control Register
movem	Move Program Memory

Mnemonic	Operation
movep	Move Peripheral Data
vsl	Viterbi Shift Left (DSP563xx, DSP566xx only)

Table 9–5: Move instructions

9.2.6 PROGRAM CONTROL INSTRUCTIONS

Mnemonic	Operation
b*cc*	Branch Conditionally (DSP563xx, DSP566xx only)
bra	Branch (DSP563xx, DSP566xx only)
brclr	Branch if Bit Clear (DSP563xx only)
brset	Branch if Bit Set (DSP563xx only)
bs*cc*	Branch to Subroutine Conditionally (DSP563xx, DSP566xx only)
bsr	Branch to Subroutine (PC relative) (DSP563xx, DSP566xx only)
bsclr	Branch to Subroutine if Bit Clear (DSP563xx only)
bsset	Branch to Subroutine if Bit Set (DSP563xx only)
debug	Enter Debug Mode
debug*cc*	Enter Debug Mode Conditionally
if*cc*	Execute Conditionally (DSP563xx, DSP566xx only)
if*cc***.U**	Execute Conditionally and Update CCR (DSP563xx, DSP566xx only)
illegal	Illegal Instruction
j*cc*	Jump Conditionally
jmp	Jump
jclr	Jump if Bit Clear (DSP563xx, DSP566xx only)
jset	Jump if Bit Set (DSP563xx, DSP566xx only)
js*cc*	Jump to Subroutine Conditionally
jsr	Jump to Subroutine
jsclr	Jump to Subroutine if Bit Clear
jsset	Jump to Subroutine if Bit Set
nop	No Operation

Mnemonic	Operation
plock	Lock Program Cache Sector (DSP563xx only)
punlock	Unlock Program Cache Sector (DSP563xx only)
punlockr	Unlock PC–Related Program Cache Sector (DSP563xx only)
pfree	Unlock all Program Cache Locked Sectors (DSP563xx only)
pflush	Reset Program Cache State (DSP563xx only)
pflushun	Reset Program Cache State to all Unlocked Sectors (DSP563xx only)
rep	Repeat Next Instruction
reset	Reset On–Chip Peripheral Devices
rti	Return from Interrupt
rts	Return from Subroutine
stop	Stop Processing (Low–Power Standby)
swi	Software Interrupt (DSP5600x only)
trap*cc*	Trap Conditionally (DSP563xx, DSP566xx only)
trap	Trap Always (DSP563xx, DSP566xx only)
wait	Wait for Interrupt (Low–Power Standby)

Table 9–6: Program control instructions

CHAPTER

LINKER

10

TASKING

CHAPTER 10

10.1 OVERVIEW

This section gives a global overview of the process of linking programs for the DSP56xxx and its derivatives. The linker executable name for the DSP5600x is **lk56**, the linker executable name for the DSP563xx/DSP566xx is **lk563**. The invocations and examples are given for the DSP563xx.

Unless explicitly stated otherwise, all invocations and examples are also valid for the other executables. Just replace **lk563** with the appropriate executable name.

The linker combines relocatable object files, generated by the assembler, or object modules in CLAS COFF object file format into one new relocatable object file (preferred extension `.out`). This file may be used as input in subsequent linker calls: the linkage process may be incremental. Normally the linker complains about unresolved external references. With incremental linking it is normal to have unresolved references in the output file. Incremental linking must be selected separately.

The linker can read normal object files and libraries of object modules and also Motorola CLAS COFF formatted object files. Modules in a library are included only when they are referenced. At the end of the linkage process the generated object, without unresolved references, will be called: a load module.

For linking Motorola CLAS objects, see the section *Linking CLAS COFF Objects*

The DSP56xxx linker is an overlaying linker. The compiler generates overlayable sections. An overlayable section contains space reservations for variables which, at C level, are local to a function. If functions do not call each other, their local variables can be overlayed in memory. It is a task of the linker to combine function call information into a call graph and to determine upon the structure of this call graph how sections can be overlayed, using the smallest amount of RAM.

Incremental linkage disables overlaying, so the last link phase should not be incremental, even if the incremental phase resolves all externals.

The following diagram shows the input files and output files of the linker:

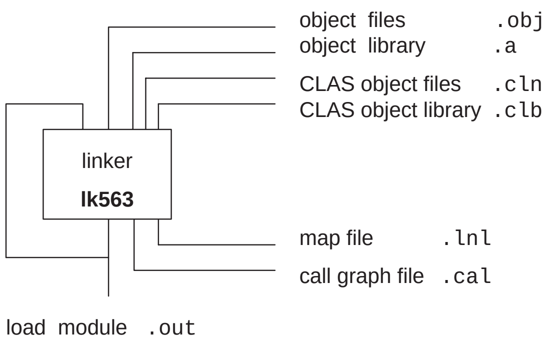


Figure 10–1: DSP563xx/DSP566xx Linker

10.2 LINKER INVOCATION

The invocation of the DSP563xx/DSP566xx linker is (use **lk56** for the DSP5600x family):

lk563 [*option*]... *file* ...

When you use a UNIX shell (**C–shell, Bourne shell**), options containing special characters (such as '**( )**') must be enclosed with **" "**. The invocations for UNIX and PC are the same, except for the **–?** option in the **C–shell**:

```
lk563  "-?"        or    lk563  -\?
```

Options may appear in any order. Options start with a '**–**'. Only the **–l***x* option is position dependent. Option may be combined: **–rM** is equal to **–r –M**. Options that require a filename or a string may be separated by a space or not: **–o***name* is equal to **–o** *name*.

file can be any object file (`.obj`), CLAS object file, object libraries (`.a`) or incrementally linker (`.out`) files. The files are linked in the same order as they appear on the command line.

The linker recognizes the following options:

Option	Description
–C	Link case insensitive (default case sensitive)
–H or **–?**	Display invocation syntax
–L *directory*	Additional search path for system libraries
–L	Skip system library search
–M	Produce a linker map (`.lnl`)
–N	Turn off overlaying (**lk56** only)
–O *name*	Specify basename of the resulting map files
–V	Display version header only
–c	Produce a separate call graph file (`.cal`)
–d *file*	Read description file information from *file*, '–' means `stdin`
–e	Clean up if erroneous result
–err	Redirect error messages to error file (`.elk`)
–f *file*	Read command line information from *file*, '–' means `stdin`
–l *x*	Search also in system library `lib`*x*`.a`
–m	Merge option for different object formats
–o *filename*	Specify name of output file
–r	Suppress undefined symbol diagnostics
–u *symbol*	Enter *symbol* as undefined in the symbol table
–v or **–t**	Verbose option. Print name of each file as it is processed
–w *n*	Suppress messages above warning level *n*.

Table 10–1: Options summary

10.2.1 DETAILED DESCRIPTION OF LINKER OPTIONS

With options that can be set from within EDE, you will find a mouse icon that describes the corresponding action.

-?/-H

Option:

–?
–H

Description:

Display an explanation of the invocation syntax at `stdout`.

Example:

```
lk563  –H
```

-C

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`. Disable the `Link case sensitive` check box.

–C

Default:

Case sensitive

Description:

With this option the linker links case insensitive. The default is case sensitive linking.

Example:

To switch to case insensitive mode, enter:

```
lk563 –C test.obj
```

Using the control program:

```
cc563 –Wlk–C test.obj
```

-c

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Enable the `Generate a separate function call graph file (.cal)` check box.

–c

Description:

Generate separate call graph file (`.cal`).

Example:

To create a call graph file (`test.cal`), enter:

```
lk563 -c test.obj
```

Using the control program:

```
cc563 -Wlk-c test.obj
```

Section *Linker Output*.

–d

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Control File`.
From the `Target` list, select a predefined target or select `User supplied target definition` and specify a target name, select the `Use project specific linker/locator control file (.dsc)` radio button and enter the name of the description file in the field below.

–d *file*

Arguments:

A filename to read description file information from. If *file* is a '**–**', the information is read from standard input.

Description:

Read description file information from *file* instead of a `.dsc` file.

Example:

To read description file information from file `56302evm.dsc`, enter:

```
lk563 –d56302evm.dsc test.obj
```

Using the control program:

```
cc56 56302evm.dsc test.obj
```

-err

Option:

In EDE this option is not useful.

–err

Description:

The linker redirects error messages to a file with the same basename as the output file and the extension `.elk`. The default filename is `a.elk`.

Example:

To write errors to the file `a.elk` instead of `stderr`, enter:

```
lk563 -err test.obj
```

To write errors to the file `test.elk` instead of `stderr`, enter:

```
lk563 -err test.obj -otest.out
```

–e

Option:

EDE always removes the output files when errors occur.

–e

Description:

Remove all link products such as temporary files, the resulting output file and the map file, in case an error occurred.

Example:

```
lk563 –e test.obj
```

–f

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Add the option to the `Additional options` field.

–f *file*

Arguments:

A filename for command line processing. The filename ”–” may be used to denote standard input.

Description:

Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.

More than one **–f** option is allowed.

Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.

2. To include whitespace in the argument, surround the argument with either single or double quotes.

3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:

   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.

   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

Example:

```
”This has a single quote ’ embedded”
```

or

```
’This has a double quote ” embedded’
```

or

```
’This has a double quote ” and \
a single quote ’”’ embedded”
```

4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non–quoted arguments, all whitespace on the next line will be stripped.

   Example:

```
”This is a continuation \
line”
     -> ”This is a continuation line”

control(file1(mode,type),\
     file2(type))
     ->
control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

Example:

Suppose the file **mycmds** contains the following line:

```
–err
test.obj
```

The command line can now be:

```
lk563 –f mycmds
```

–L

Option:

Select the `Project | Directories...` menu item. Add one or more directory paths to the `Library Files Path` field.

–L [*directory*]

Arguments:

The name of the directory to search for system libraries.

Description:

Add *directory* to the list of directories that are searched for system libraries. Directories specified with **–L** are searched before the standard directories specified by the environment variable C563LIB (for the DSP563xx/DSP566xx, C56LIB for the DSP5600x). If you specify **–L** without a directory, the environment variable C563LIB (or C56LIB) is not searched for system libraries. You can use the **–L** option more than once to add several directories to the search path for system libraries. The search path is created in the same order as in which the directories are specified on the command line.

Example:

```
lk563 -Lc:\c563\lib\563xx test.obj
```

–l

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Add the option to the `Additional options` field.

–l *x*

Arguments:

A string to form the name of the system library `lib`*`x`*`.a`.

Description:

Search also in system library `lib`*`x`*`.a`, where *x* is a string. The linker first searches for system libraries in any directories specified with **–L***directory*, then in the standard directories specified with the environment variable C563LIB (or C56LIB), unless the **–L** option is used without a directory specified.

This option is position dependent (see section *Linking with Libraries*).

Example:

To search in the system library `libc16.a` after the user object and library are linked, enter:

```
lk563 myobj.obj mylib.a –lc16
```

-M

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Enable the `Generate a linker listing file (.lnl)` check box.

–M

Description:

Produce a linker map file (`.lnl`). If no output filename is specified the default name is `a.lnl`.

Example:

To create the map file `a.lnl`, enter:

```
lk563 –M test.obj
```

Section *Linker Output*,
–O.

-m

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Add the option to the `Additional options` field.

–m

Description:

Merge option to solve incompatibilities between different object formats.

-N (lk56 only)

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`. Add the option to the `Additional options` field.

–N

Description:

Turn off overlaying. This can be useful for debugging.

-O

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Add the option to the `Additional options` field.

–O *name*

Arguments:

The basename to be used for map files.

Description:

Use *name* as the default basename for the resulting map files.

Example:

To create the map file `test.lnl` using the linker, enter:

```
lk563 –M –Otest test.obj
```

Using the control program:

```
cc563 –Wlk–M –Wlk–Otest test.obj
```

Section *Linker Output*,
–M.

-o

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Add the option to the `Additional options` field.

–o *filename*

Arguments:

An output filename.

Default:

`a.out`

Description:

Use *filename* as output filename of the linker. If this option is omitted, the default filename is `a.out`.

Example:

To create the output file `test.out` instead of `a.out`, enter:

```
lk563 test.obj –otest.out
```

–r

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Add the option to the `Additional options` field.

–r

Description:

Specify incremental linking. No report is made for unresolved symbols, and the function overlaying is disabled.

Section *Linker Output*.

-u

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Add the option to the `Additional options` field.

–u *symbol*

Arguments:

The name of a symbol to undefine.

Description:

Enter *symbol* as undefined in the symbol table. This is useful for linking from a library.

Example:

To force symbol `main` as undefined, enter:

```
lk563 –u main mylib.a
```

Section *Linking with Libraries*.

–V

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Add the option to the `Additional options` field.

–V

Description:

With this option you can display the version header of the linker. This option must be the only argument of **lk563**. Other options are ignored. The linker exits after displaying the version header.

Example:

```
lk56 -V

TASKING DSP5600x C linker        vx.yrz Build nnn
Copyright 1995-year Altium BV    Serial# 00000000
```

```
lk563 -V

TASKING DSP563xx/6xx linker      vx.yrz Build nnn
Copyright 1996-year Altium BV    Serial# 00000000
```

–v

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Enable the `Print the name of each file as it is processed` check box.

–v
–t

Description:

Verbose option. Print the name of each file as it is processed. Also shows which objects are extracted from libraries.

Example:

`lk563 –v test.obj`

```
lk563 V008 (1): Embedded environment \c563\etc\def_targ.dsc
                read, relaxed addressing mode check enabled
lk563 V003 (1): Starting pass 1
lk563 V002 (1): File currently in progress:
                test.obj
lk563 E208 (0): Found unresolved external(s):
                _printf – (test.obj)
                __START – (test.obj)
lk563 V003 (1): Starting pass 2
lk563 V002 (1): File currently in progress:
                test.obj
lk563 V005 (1): Removing file .\CD5668a.tld
```

Using the control program:

`cc563 –Wlk–v test.obj`

–w

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Linker Miscellaneous`.
Select a warning level from the `Suppress warning messages above` list box.

–w *level*

Arguments:

A warning level between 0 and 9 (inclusive).

Default:

–w8

Description:

Give a warning level between 0 and 9 (inclusive). All warnings with a level above *level* are suppressed. The level of a message is printed between parentheses after the warning number. If you do not use the **–w** option, the default warning level is 8.

Example:

To suppresses warnings above level 5, enter:

```
lk563 –w5 test.obj
```

Using the control program:

```
cc563 –Wlk–w5 test.obj
```

Section *Type Checking*.

10.3 LIBRARIES

There are two kinds of libraries. One of them is the user library. If you make your own library of object modules, this library must be specified as an ordinary filename. The linker will not use any search path to find such a library. The file must have the extension `.a` or `.clb`. Example:

```
lk563 start.obj –fobj.lnk mylib.a
```

or, if the library resides in a sub directory:

```
lk563 start.obj –fobj.lnk libs\mylib.a       (PC)
lk563 start.obj –fobj.lnk libs/mylib.a       (UNIX)
```

The other kind of library is the system library. You must define system libraries with the **–l** option. With the option **–lcm** you specify the system library `libcm.a`.

10.3.1 LIBRARY SEARCH PATH

The linker searches for system library files according to the following algorithm:

1. Use the directories specified with the **–L***directory* options, in a left–to–right order. For example:

   PC:

   ```
   lk563 –L..\lib\563xx –L\usr\local\lib start.obj
   –fobj.lnk –lc24
   ```

   UNIX:

   ```
   lk563 –L../lib/563xx –L/usr/local/lib start.obj
   –fobj.lnk –lc24
   ```

2. If the **–L** option is not specified without a directory, check if the environment variable C563LIB exists. If it does, use the contents as a directory specifier for library files. It is possible to specify more than one directory in the environment variable C563LIB by separating the directories with a directory separator. Valid directory separators are:

   PC:

   ;   **,**   *space*

UNIX:

:   ;   ,   *space*

Instead of using **–L** as in the example above, the same directory can be specified using C563LIB:

PC:

```
set C563LIB=..\lib\563xx;\usr\local\lib
lk563 start.obj –fobj.lnk –lc24
```

UNIX:

if using the Bourne shell (sh), or korn shell (ksh)

```
C563LIB=../lib/563xx:/usr/local/lib
export C563LIB
lk563 start.obj –fobj.lnk –lc24
```

or if using the C–shell (csh)

```
setenv C563LIB ../lib/563xx:/usr/local/lib
lk563 start.obj –fobj.lnk –lc24
```

3. Search in the `lib` directory relative to the installation directory of **lk563** for library files.

   PC:

   lk563.exe is installed in the directory `C:\C563\BIN`
   The directory searched for the library file is `C:\C563\LIB`

   UNIX:

   lk563 is installed in the directory `/usr/local/c563/bin`
   The directory searched for the library file is `/usr/local/c563/lib`

   The linker determines run–time which directory the binary is executed from to find this `lib` directory.

4. If the library is still not found, search in the processor specific subdirectory of the `lib` directory relative to the installation directory of **lk563** for library files. For example:

   PC:

   ```
   C:\C563\LIB\563xx
   ```

UNIX:

```
/usr/local/c563/lib/563xx
```

A directory name specified with the **–L***directory* option or in C563LIB (or C56LIB for the DSP5600x) may or may not be terminated with a directory separator, because **lk563** inserts this separator, if omitted.

10.3.2 LINKING WITH LIBRARIES

If you are linking from libraries, only those objects you need are extracted from the library. This implies that if you invoke the linker like:

```
lk563 mylib.a
```

nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

It is possible to force a symbol as undefined with the option **–u**:

`lk563 –u main mylib.a` (space between **–u** and main is optional)

In this case the symbol `main` will be searched for in the library and (if found) the object containing `main` will be extracted. If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found. See also the library member search algorithm in the next section.

The position of the library is important, if you specify:

```
lk563 –lc24 myobj.obj mylib.a
```

the linker starts with searching the system library `libc24.a` without unresolved symbols, thus no module will be extracted. After that, the user object and library are linked. When finished, all symbols from the C library remain unresolved. So, the correct invocation is:

```
lk563 myobj.obj mylib.a –lc24
```

All symbols which remain unresolved after linking `myobj.obj` and `mylib.a` will be searched for in the system library `libc24.a`.

The link order for objects, user libraries and system libraries is the order in which they appear at the command line. Objects are always linked, object modules in libraries are only linked if they are needed.

10.3.3 LIBRARY MEMBER SEARCH ALGORITHM

A library built with **ar563** always contains an index part at the beginning of the library. The linker scans this index while searching for unresolved externals. However, to keep the index as small as possible, only the defined symbols of the library members are recorded in this area.

When the linker finds a symbol that matches an unresolved external, the corresponding object file is extracted from the library and is processed. After processing the object file, the remaining library index is searched. If after a complete search unresolved externals are introduced, the library will be scanned again.

The **–v** option shows how libraries have been searched and which objects have been extracted.

10.4 LINKING CLAS COFF OBJECTS

The linker **lk563** reads object files (and object libraries) in the Motorola CLAS COFF format used by the Motorola assembler. The output file of **lk563** is always in the IEEE–695 based TIOF format. Output in Motorola CLAS format is not supported.

High level language debug information represented in the CLAS object module will not be placed in the output file of the linker **lk563**. High level source debugging can only be done on modules compiled with the TASKING **c56** or **c563** compiler.

The main reason of linking Motorola CLAS object files using **lk563** is to be able to link existing object modules which are not available in source (otherwise these sources could be compiled/assembled by the TASKING compilers/assemblers).

Because of the differences in the handling of section scoping in the Motorola assembler and the TASKING assemblers, only public labels at the GLOBAL level can be matched.

Symbols in object files of the same type must match exactly.

As a result of this, it is possible that one external symbol definition matches one or more public symbols.

Overlaying is not supported for sections defined in the Motorola CLAS object format.

The Motorola section names of linked–in CLAS COFF files are converted into TASKING section names according to the following rules:

Absolute sections:	*motorola_name*+'_'+*address*
Relocatable sections:	*motorola_name*+'_'+*section_attributes*

Section attributes have the following syntax:

*access***Y***ninit***C**[**N** | **P**]

The *access* and *init* parts can be:

	access	*init*
Text sections:	X	
Data sections:	W	I
BSS sections:	W	B
Overlay sections:	W	B
Pad sections:	W	F
Block sections:	W	I

Table 10–2: Section attributes

All section type letters are explained in section *ST Command* of Appendix I, *IEEE–695 Object Format*.

An example Motorola (relocatable) section name is:

```
motsec_WY5BCN
```

10.5 LINKER OUTPUT

The linker produces an IEEE–695 object output file and, if requested, a linker map file, and/or a call graph file.

The linker output object is still relocatable. It is the task of the locator to determine the absolute addresses of the sections. The linker combines sections with the same name to one (bigger) output section.

The linker produces a map file if the option **–M** is specified. The name of the map file is the same as the name of the output file. The extension is `.lnl`. If no output filename is specified the default name is `a.lnl`. The map file is organized per linked object. Each object is divided in sections and symbols per section. The map file shows the relative position of each linked object from the start of the section.

The generated call graph will also be printed in the map file. The command line option **–c** forces the linker to generate a separate call graph file with a compressed call graph. The filename extension of this file is `.cal`.

If the linker is used for incremental linking, the **–r** option must be used. The effect is, that unresolved symbol diagnostics will not be generated, and overlaying is not done (see 10.6). In this case, the output of the linker can be used again as input object. A call graph will always be generated.

A (part of) sample map file:

```
Call graph(s)
=============

Call graph 1:

main()
  |
  +--puts()
       |
       +--fputc()
            |
            +--_flsbuf()
                 |
                 +--_write()
                 |    |
                 |    +--_iowrite()
                 |         |
                 |         +--_simo()
                 |
                 +--_iowrite()
                      |
                      +--_simo()
```

```
Pool offsets
============

Pool #1: .xovl  (Total of 103 bytes)

                Pool: .xovl
                off     siz
    puts()      0       5
   fputc()      5       1
 _flsbuf()      6       85
  _write()      91      9
_iowrite()      100     3


Object: hello.obj
=================

Section:.ptext ( Start = 0x0 )
0x00000000 E Fmain


Section:.xstring ( Start = 0x0 )

Object: puts.obj
================

Section:.ptext ( Start = 0x4 )
0x00000004 E Fputs

Object: cstart.obj
==================

Section:.ptext ( Start = 0x3b )
0x0000003b E F_START
0x0000003b E F_copytable
0x0000004d E F_exit
0x00000069 E cptable_clr
0x0000004e E cptable_copy


Section: generated0 ( Start = 0x0 )


Section: generated1 ( Start = 0x0 )
......
```

The addresses in the map file are offsets relative to the start of the section in the output file. For instance, symbol `F_START` in the object module `cstart.obj` is at offset 0x3b from the output `.ptext` section. Function `F_START` also starts at offset 0x3b from the start of the resulting `.ptext` section. The `E` after the address indicates the label is external.

10.6 OVERLAY SECTIONS

In order to make memory use in the static memory model more efficient, the compiler generates special sections, with the overlay attribute, which must be overlayed by the linker. Each C function has its own section with local variables, temporaries etc. The linker builds a call graph to determine a valid overlay of the sections of functions which do not call each other.

For example:

```
main()
{
     int j;
     printf( ”hello\n” );
     j = 2;
     foo(j);
}
foo( int j )
{
     int i;
     i = j;
}
```

The linker detects that foo does not call `printf`, and `printf` does not call `foo`. The compiler generates an overlayable data section for the local, direct addressable, variable `i`. `printf`, which also has local variables, gets its own overlayable data section. The linker puts the overlay sections of these two functions at the same (direct addressable) memory area. The advantage is that the target memory is used more efficiently.

10.7 TYPE CHECKING

10.7.1 INTRODUCTION

By default the compiler and the assembler generate high–level type information. Unless you disable generation of type information (**–g0**), each object contains type information of high–level types. The linker compares this type information and warns you if there are conflicts. The linker distinguishes four types of conflicts:

1. Type not completely specified (W109). Occurs if you do not specify the depth of an array, or if you do not specify arguments in one of the function prototypes. The linker does not report this type of conflict unless you specify a warning level 9 (**–w9**), default is warning level 8.
2. Compatible types, different definitions (W110). Occurs if for instance you link a short with an int. The DSP566xx takes both as 16 bits, so there will not be a problem. However, the code is not portable. Also structures or types with different names produce this warning. The warning level for this message is 8, so you can switch off this kind of message by specifying warning level 7 or less (**–w7**).
3. Signed/unsigned conflict (W111). If you link a signed int with an unsigned int, you get this message. In many cases there will be no problem, but the unsigned version can hold a bigger integer. The warning level of this warning is 6 and can be suppressed by specifying a warning level of 5 or less (**–w5**).
4. All other type conflicts (W112). If you get warning 112, there is probably a more serious type conflict. This can be a conflict in a function return type, a conflict in length between two built in types (short/long) or a completely different type. This warning has a level of 4, and can be switched off with warning level 3 or less (**–w3**).

10.7.2 RECURSIVE TYPE CHECKING

The linker compares types recursively. For instance, the type of `foo`:

```
struct s1 {
     struct s2 *s2_ptr;
};

struct s2 {
     int  count;
} sample;

struct s1 foo = { &sample };
```

If you compile this source and link it with another compiled source with only `struct s2` different:

```
struct s1 {
     struct s2 *s2_ptr;
};

struct s2 {
     short count;
};

extern struct s1 foo;
```

message W112 (type conflict) will be generated. Although `struct s1` is the same in both cases, this is a real type conflict: For instance, the code `”foo.s2_ptr->count++”` produces different code in both objects.

If you have several conflicts in one symbol, the linker reports only the one with the lowest warning level. (The most serious one.)

10.7.3 TYPE CHECKING BETWEEN FUNCTIONS

If you use K&R style functions, it is not possible to check the type of the arguments and the number of arguments. Return types are ’int’ if not specified. Prototypes are only needed if a function has a non–integer return type:

```
test2( par )
int par;
{
     test1( par );
     return test3( 1, 2 );
}
```

In this case, `test1` (defined in another source) has a return type `void`, and `test3` has a return type `int`, which is the default. At the default warning level, the linker does not report any conflict. If you should specify warning level 9 (**–w9**), the linker reports a 'not completely specified' type, because the linker is not able to check the arguments. Conflicts in return types cause real type conflicts at warning level 4.

If the source is ANSI style (which is recommended), the linker checks the types of all parameters, and the number of parameters. In this case the source of the example above looks like:

```
void test1( int );   /* ANSI style prototypes */
int  test3( int, int );

test2( int par ) /* ANSI style function definition*/
{
     test1( par );
     return test3( 1, 2 );
}
```

Another source, containing the definition of `test1` and `test3` may look like:

```
void test1( int one )
{
     /*
     **  code for function test1
     */
     .
     .
     .
}
```

```
int test3( int one, int two )
{
     /*
     **  Code for function test3
     */
     .
     .
     .
}
```

Prototypes are only needed for functions which are referenced before they are defined within one source. However, it is a good practice to include a prototype file with prototypes of all the functions in a file. If you do so, type checking for functions is done by the compiler. Nevertheless, if you do not compile all sources after you have changed the prototype file, the linker will report the type conflict.

It is possible to add ANSI style prototypes to K&R style C–code. In this case full type checking for functions becomes available. To accomplish this, make a new header file with all prototypes for all functions in your application. Include this file in each source, or tell the compiler to include it for you by means of the option **–H**:

```
cc563 –c –Hproto.h *.c
```

10.7.4 MISSING TYPES

In C you are allowed to define pointers to unspecified objects. The linker is not able to check such types. For instance:

```
struct s1 {
     struct s2 *s2_ptr;
};

struct s1 foo;
```

The structure `s2` is not specified. Because the linker is not able to check whether `struct s2` is the same in all sources, a warning at level 9 will be generated:

lk563 W102 (9) <*name*>: Incomplete type specification, type index = T101

It is possible that the struct s2 is known in an other source. If this source uses variable `foo`, a second message is generated, reporting a level 9 type conflict:

lk563 W109 (9) <f1>: Type not completely specified for symbol <foo> in <f2>

Because the type definition is not complete, the first warning reports that the linker cannot check the type, although this is allowed in C. This message is given once for each object for each incomplete type. The second warning reports a difference in types, an incomplete type versus a complete type.

All these warnings are only generated if you specify warning level 9 (**–w9**).

10.8 LINKER MESSAGES

There are four kinds of messages: fatal messages, error messages, warning messages and verbose messages. Fatal messages are generated if the linker is not able to perform its task due to the severity of the error. In those situations, the exit code will be 2. Error messages will be reported if an error occurred which is not fatal for the linker. However, the output of the linker is not usable. The exit code in case of one or more error messages will be 1. Warning messages are generated if the linker detects potential errors, but the linker is unable to judge those errors. The exit code will be 0 in this case, indicating a usable `.out` file. Of course, if the linker reports no messages at all, the exit code is 0 also.

Each linker message has a built–in warning level. With option **–w***x* it is possible to suppress messages with a warning level above *x*.

Verbose messages are generated only if the verbose option (**–v**) is on. They report the progress of the link process.

Linker messages have the following layout:

```
DSP563xx/6xx object linker vx.y rz    SN00000000-000 (c) year TASKING, Inc.
lk563 W112 (4) a.o: Type conflict for symbol <f> in b.o
```

The first line shows the banner of the DSP56xxx linker. The second line reports a type conflict in the file `a.obj`. Apparently there is a conflicting type definition of the function `f` in module `b.obj`. The number between parentheses after the warning number, '(4)', shows the warning level.

There are four message groups:

1. **Fatal** (always level 0):
   - Write error
   - Out of memory
   - Illegal input object
2. **Error** (always level 0):
   - Unresolved symbols (and no incremental linking)
   - Can't open input file
   - Illegal recursive use of an non reentrant function
3. **Warning** (levels from 1 to 9):
   - Type conflict between two symbols
   - Illegal option (Ignored)
   - No system library search path, and system library requested
4. **Verbose** (level not relevant, only given with option **–v**):
   - Extracting files from a library
   - Current file/library name
   - Pass one or pass two
   - Rescanning library for new unresolved symbols
   - Cleaning up temp files
   - warning level

CHAPTER

LOCATOR

11

TASKING

CHAPTER 11

11.1 OVERVIEW

This chapter describes the locator and its control language for the DSP5600x (**lc56**) and DSP563xx/DSP566xx (**lc563**). **lc563** is used to describe all three locators, unless explicitly stated otherwise.

The task of the locator is to locate a `.out` file, made by **lk563**, to absolute addresses. In an embedded environment an accurate description of available memory and information about controlling the behavior of the locator is crucial for a successful application. For example, it may be necessary to port applications to processors with different memory configurations, or it may be necessary to tune the location of sections to take full advantage of fast memory chips. To perform its task the locator needs a description of the derivative of the DSP56xxx used. The locator uses a special language for this description: DELFEE, which stands for DEscriptive Language For Embedded Environments. This control language is used in a special file, which is called the description file. See Appendix G *DEscriptive Language For Embedded Environments* for detailed information.

The description file is an optional parameter in the locator invocation. Without a description file name on the command line, or without the **–d** option, the locator searches the file `def_targ.dsc` in the current directory or in directory `etc` in the product tree.

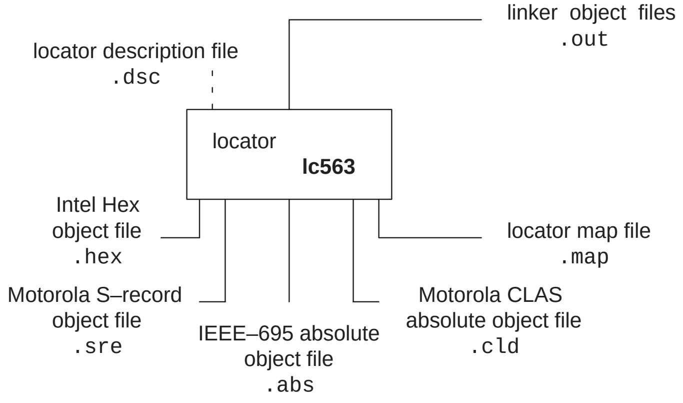


Figure 11–1: Locator

11.2 INVOCATION

The invocation of the locator is:

lc563 [*option*]... [*file*]...

When you use a UNIX shell (**C–shell, Bourne shell**), options containing special characters (such as '**( )**') must be enclosed with **" "**. The invocations for UNIX and PC are the same, except for the **–?** option in the **C–shell**:

lc563 "–?" or **lc563 –\?**

Options may appear in any order. Options start with a '**–**'. They may be combined: **–eM** is equal to **–e –M**. Options that require a filename or a string may be separated by a space or not: **–o***name* is equal to **–o** *name*. *file* may be any file with a `.out` or `.dsc` extension.

The locator recognizes the following options:

Option	Description
–H or **–?**	Display invocation syntax
–M[*n*]	Produce a locate map file (`.map`) with maximum width *n* (*n* > 132)
–S *space*	Generate specific *space*
–V	Display version header only
–c	Don't generate ROM copy for re–initializing data memory
–d *file*	Read description file information from *file*, '–' means `stdin`
–e	Clean up if erroneous result
–em*macro*[**=***def*]	Define preprocessor *macro*
–err	Redirect error messages (`.elc`)
–f *file*	Read command line information from *file*, '–' means `stdin`
–f *format*	Specify output format
–o *filename*	Specify name of output file
–p	Make a proposal for a software part on `stdout`
–r	Romable application
–s	Strip debug info from the input

Option	Description
–v	Verbose option. Print name of each file as it is processed
–w *n*	Suppress messages above warning level *n*.

Table 11–1: Options summary

11.2.1 DETAILED DESCRIPTION OF LOCATOR OPTIONS

With options that can be set from within EDE, you will find a mouse icon that describes the corresponding action.

–?/-H

Option:

–?
–H

Description:

Display an explanation of options at stdout.

Example:

```
lc563  –?
```

–c

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`. Disable the `Generate copy table for re–initializing data memory` check box.

–c

Description:

Do not generate ROM–copy for re–initializing data memory. In some special situations target memory usage can be minimized by not generating a ROM copy of the initialized data variables in program memory. The drawback of using this option is that program initialization is done during object image loading. So, restarting the program requires downloading the whole program again. Also, when the program runs from an EPROM, no attempt is made to clear or fill data sections, so a program depending on cleared or initialized RAM variables will fail then.

Here is a C example to demonstrate this:

```
/* NOTE: the copy table is handled by the startup code
   just before calling main() */

int flag = 1;

void main(void)
{
    flag = 0;
}
```

The `flag` global variable will read 1 on the first run and 0 on all subsequent runs, because it has been reset in `main()` and is never restored unless the program is downloaded again.

Normally, when running from an EPROM, the startup code will take care of clearing and initializing RAM data memory by using a ROM–copy of the initialized data variables. During program startup this ROM–copy is copied from program memory to data memory, thus assuring the program will have the same initial values on every program restart. The used bootstrap loader only needs to be able to fill program memory.

Using the **–c** option will initialize the object image with all required data values directly. So, when loading the object image, no additional copy action is required. This will save the memory space for the ROM copy at the cost of not being able to restart the program with the same initial values. Every program restart requires downloading the object image first.

The **–c** option is of great use when loading the object image from a host processor, because the program code and data can be loaded into a target system with the exact required memory space to run correctly (no extra space for additional variable images in ROM is required). To allow this, the bootstrap loader must be capable of handling data for different memory spaces.

To get a maximum decrease in memory usage, make sure to compile your C startup code without the code for initializing the data memory. Otherwise the locator will generate a copy table with zero entries to prevent unresolved references. For the Motorola DSP56xxx toolchain this means compiling the startup code with the define NOCOPY set.

–d

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Control File`.
From the `Target` list, select a predefined target or select `User supplied target definition` and specify a target name, select the `Use project specific linker/locator control file (.dsc)` radio button and enter the name of the description file in the field below.

–d *file*

Arguments:

A filename to read description file information from. If *file* is a '**–**', the information is read from standard input.

Description:

Read description file information from *file* instead of a `.dsc` file.

Example:

To read description file information from file `56302evm.dsc`), enter:

```
lc563 -d56302evm.dsc test.out
```

–e

Option:

EDE always removes the output files on errors.

–e

Description:

Remove all locate products such as temporary files, the resulting output file and the map file, in case an error occurred.

Example:

```
lc563 –e test.out
```

-em

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`.
Add the option to the `Additional options` field.

–em*macro*[**=***def*]

Arguments:

The macro you want to define and optionally its definition.

Description:

Define *macro* to the preprocessor, as in #define. If *def* is not given ('**=**' is absent), '1' is assumed. Any number of symbols can be defined. The definition can be tested by the preprocessor with #if, #ifdef and #ifndef, for conditional locating. If the command line is getting longer than the limit of the operating system used, the **–f** option is needed.

Example:

```
lc563 myproject.out –o myproject.abs –emEDE=\"myproject.i\" –M
```

-err

Option:

In EDE this option is not useful.

–err

Description:

The locator redirects error messages to a file with the same basename as the output file and the extension `.elc`. The default filename is `a.elc`.

Example:

To write errors to the file `a.elc` instead of `stderr`, enter:

```
lc563 –err test.out
```

To write errors to the file `test.elc` instead of `stderr`, enter:

```
lc563 –err test.out –otest.abs
```

–f

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`.
Add the option to the `Additional options` field.

–f *file*

Arguments:

A filename for command line processing. The filename ”–” may be used to denote standard input.

file may not be a number in the range 0–4, because these numbers are used to specify an output format.

Description:

Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.

More than one **–f** option is allowed.

Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.

2. To include whitespace in the argument, surround the argument with either single or double quotes.

3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:

   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.

b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

Example:

```
”This has a single quote ’ embedded”
```

or

```
’This has a double quote ” embedded’
```

or

```
’This has a double quote ” and \
a single quote ’”’ embedded”
```

4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non–quoted arguments, all whitespace on the next line will be stripped.

   Example:

```
”This is a continuation \
line”
     -> ”This is a continuation line”

control(file1(mode,type),\
     file2(type))
     -> control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

Example:

Suppose the file `mycmds` contains the following line:

```
–err
test.out
```

The command line can now be:

```
lc563 –f mycmds
```

-f format

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Output Format`.
Select one of the output formats.

–f *format*

Arguments:

format can be one of the following output formats:

0	=	TIOF 695
1	=	IEEE Std. 695 (Default)
2	=	Motorola S–record
3	=	Intel Hex
4	=	Motorola CLAS

Description:

Specify an output format. The default output format is IEEE Std. 695 (**–f1**), which can directly be used by the CrossView Pro debugger. The other output formats can be used for loading into a PROM–programmer.

Section *Format Suboptions*,
Appendix I, *IEEE–695 Object Format*,
Appendix J, *Motorola S–Records*,
Appendix K, *Intel Hex Records*.

–M

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`. Enable the `Produce a memory map file (.map)` check box.

–M[*n*]

Arguments:

Optionally the maximum line width ($n > 132$). If you omit the width, the default is 132 characters.

Description:

Produce a locate map (`.map`). If no output filename is specified the default name is `a.map`. The map file shows the absolute position of each section. External symbols are listed with their absolute address, both sorted on address and sorted on symbol.

Example:

To create the map file `a.map`, enter:

```
lc563 –M test.out
```

-o

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`.
Add the option to the `Additional options` field.

-o *filename*

Arguments:

An output filename.

Default:

The default filename depends on the output format specified:

Format	Default output name
0	`a.abs`
1	`a.abs`
2	`a.sre`
3	`a.hex`
4	`a.cld`

Description:

Use *filename* as output filename of the locator.

Example:

To create the output file `test.abs` instead of `a.abs`, enter:

```
lc563 test.out -otest.abs
```

-p

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`.
Add the option to the `Additional options` field.

-p

Description:

Make a proposal for a software part in a description file on standard output.

–r

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`.
Add the option to the `Additional options` field.

–r

Description:

By default locator generated sections, like the copy table, are writable, so they can be placed in RAM. Setting the **–r** option makes them read–only, which is reuired for a romable application.

-S

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`.
Select a code space from the `Generate code for space` list box.

–S *space*

Arguments:

The name of a space from a `.dsc` file.

Default:

For the IEEE–695 format, by default code is generated for all spaces.

Description:

With this option you can generate a specific output file for a specified *space* instead of generating an output file containing all spaces.

For the Motorola S–record and Intel Hex format, the locator can only generate records for one space at a time, because the formats do not make a distinction between memory spaces.

Example:

To generate code for space `P` instead of all spaces, enter:

```
lc563 test.out –S P
```

-s

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`. Disable the `Include symbolic debug information` check box.

–s

Description:

Strip debug information from the input file.

-V

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`. Add the option to the `Additional options` field.

–V

Description:

With this option you can display the version header of the locator. This option must be the only argument of **lc563**. Other options are ignored. The locator exits after displaying the version header.

Example:

```
lc56 –V

TASKING DSP5600x C linker       vx.yrz Build nnn
Copyright 1995–year Altium BV   Serial# 00000000

lc563 –V

TASKING DSP563xx/6xx linker     vx.yrz Build nnn
Copyright 1996–year Altium BV   Serial# 00000000
```

-v

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`. Enable the `Print the name of each file as it is processed` check box.

-v

Description:

Verbose option. Print the name of each file as it is processed.

Example:

`lc563 -v test.out`

```
  lc563 V001 (1): Output format: IEEE 695
  lc563 V004 (1): Warning level 8
  lc563 V007 (1): Found file <def_targ.dsc> via path
                  \c563\etc\def_targ.dsc
  lc563 V002 (1): Starting pass 1
  lc563 V000 (1): File currently in progress:
                  test.out
```

–w

Option:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Locator Miscellaneous`.
Select a warning level from the `Suppress warning messages above` list box.

–w *level*

Arguments:

A warning level between 0 and 9 (inclusive).

Default:

–w8

Description:

Give a warning level between 0 and 9 (inclusive). All warnings with a level above *level* are suppressed. The level of a message is printed between parentheses after the warning number. If you do not use the **–w** option, the default warning level is 8.

Example:

To suppresses warnings above level 5, enter:

```
lc563 –w5 test.out
```

Using the control program:

```
cc563 –Wlc–w5 test.out
```

11.2.2 FORMAT SUBOPTIONS

The layout of the **–f***format* switch as shown in the previous section has some extra capabilities. The general form of *format* is:

format_number[*format_option*]...

The first character is a single digit known as the format specifier. This format specifier can be followed by one or more *format_options*. These *format_options* are in principle output format dependent. Currently the following *format_options* are known:

Suboption	Description	Valid Formats
a	Sort addresses in ascending order	2
b*size*	Specify the output buffer size	1, 2, 3
c	Generate Intel Hex file for each chip	2, 3
s	Emit start address record	3
S1, **S2**, **S3**	Force Motorola S1, S2 or S3 records	2
u	Unsorted addresses (default)	2

Table 11–2: Format suboptions

For format 2 and 3 the buffer size sets the length of an output record exclusive record code, address and checksum. The following Intel Hex record has a buffer size of 32 bytes:

```
:20004000                               ( Record code and address )
000028A101002925FBFF6015FFFF6B25DDFF0010000000000000020A101002925
                                                    (32 data bytes)
8F                                                     ( Checksum )
```

For format 1, *size* is the maximum number of bytes in one LD command.

Examples:

Format 2 with buffer size 64:	**–f2b64**
Format 1 with buffer size 128:	**–f1b128**
Format 2 with ascending addresses:	**–f2a**
Format 2 with unsorted addresses:	**–f2u**
Format 3 with separate hex files for each chip:	**–f3c**
Format 3 including start address record:	**–f3s**
Format 2 with S2 records:	**–f2S2**

11.3 GETTING STARTED

The locator invocation is normally done via EDE or the control program. In this section you will invoke the locator as a separate tool in order to get a better understanding of the use of options and the description file.

You can find a more detailed description of the descriptive language for embedded environments (DELFEE) in Appendix G *DEscriptive Language For Embedded Environments*.

If you want to locate the `calc` demo, you need the relocatable demo file `calc.out` as input for the locator. You can generate this file by copying the contents of the directory `examples\asm` (`examples/asm` for UNIX) to your working directory, and invoke the control program:

```
cc563 –cl startup.asm calc.asm –o calc.out
```

Be sure that the `bin` directory of the DSP56xxx tools is in the search path. The option **–cl** tells the control program to stop after linking and to suppress the locating phase. The file you made by this command is the complete demo, but still in a relocatable form. Now, you can locate this relocatable file `calc.out` to absolute addresses by typing:

```
lc563 –M calc.out
```

The **–M** option causes **lc563** to make a map file. The default output file format is IEEE–695 (**–f1** option). Since you did not specify an output name, the default output name `a.abs` will be generated. (For **–f0** and **–f1** the default is `a.abs`, for **–f2** the default is `a.sre`, for **–f3** the default is `a.hex` and for **–f4** the default is `a.cld`.) After the invocation, the locator has generated two files:

– `a.abs`, The IEEE 695 output file
– `a.map`, The locate map file

If you want to give the output file a specific name, you must use the **–o***file* option:

```
lc563 –M calc.out –o calc.abs
```

You may need to adjust the description file. In a description file you can change the locating algorithm of the locator. If you do not specify a description file, the locator uses the file `def_targ.dsc` from the `etc` sub directory (in the DSP56xxx product tree). If you do not want to change this original description file (which is advisable), make a copy of file `def_targ.dsc` to your working directory.

You can change the copy of the description file. Everything after a comment (`//`) until the end of the line is ignored. As an example, change the lines:

```
amode P_far {
     section selection=x;
     section selection=r;
     table;
     copy;
}
```

into:

```
amode P_far {
     section .text;
     section .ptext;
     table;
     copy;
}
```

The effect will be that the location order of the sections `.text` and `.ptext` is now forced to be fixed.

Locate again to see the effect. The modified description file `def_targ.dsc` in your working directory will be found before the original version in the `etc` directory. Because you want to compare the map files, choose another output name:

```
lc563 –M calc.out –ocalc_o.abs
```

Now you can compare `calc.map` and `calc_o.map`.

If you want to choose between a description file with and without the changes you made, you must rename the `def_targ.dsc` in your working directory to, for example, `order.dsc`. If you want the changed version of the description, you can invoke the locator as follows:

```
lc563 –M –d order calc.out –ocalc_o.abs
```

The space between **–d** and **order** is optional. If you do install `order.dsc` in the `etc` subdirectory, you can use the option **–dorder** from any working directory.

If you want to know more about the locate language DELFEE, read Appendix G.

11.4 LOCATOR TARGET BOARD SUPPORT

The locator description files that are supplied with the tool set contain conditional parts to place the stack in the desired memory space. The control variables for this selection are automatically generated when compiling with the control program **cc563** or from EDE, and are set to the normal defaults when no memory space options are supplied.

To set the stack size and the heap size, all description files contain `STACKSIZE` and `HEAPSIZE` defines that can be set from the locator command line (with **–em**) or from EDE. Default values that are supplied are 0.75k (or 1k for non–L–stack models) stack and 4k heap; some smaller targets have other appropriate values. See the description file that is concerned.

A problem that arises with the stack in L–memory is that L–memory is allocated first, and only after that X– and Y–memory. Therefore L–memory takes precedence, and the stack area will normally block the allocation of near X– and Y–variables. A define `NEARXYSIZE` is provided in the description files to reserve space for such variables; it is set to zero if not defined externally.

The `.cpu` files have defines for the memory switch and the cache size (`SWITCH_SELECT` and `CACHE_SELECT`) that can be set from the locator command line. These defines are set to default values if they are not specified on the command line.

An overview of the defines is given in the following table.

Locator control	Default	Description
STACKSIZE	768 or 1K	Stack size (bytes)
HEAPSIZE	4K	Heap size (bytes)
NEARXYSIZE	0	Area size for near X/Y variables
CACHE_SELECT	0 or 1	Enable/disable cache
SWITCH_SELECT	0 or 1	Enable/disable memory switch

Table 11–3: Locator controls

11.5 FORCE CONST SECTIONS

Previous toolchain versions handled initialized P memory data sections incorrectly, and wasted a lot of P memory on intializers for constant data sections. Two modifiers have been added to the **org** statement, to define constant (`const`) and initialized (`int`) data sections. The `const` data sections that were previously generated (for string literals only) were named ".xstring" (or any other memory), but these were still copied from the copy table. The new `const` data sections are named ".xconst" etc., and they are not copied anymore; instead, the debugger places them in the appropriate memory directly. This saves a lot of memory in applications that use large tables or a lot of strings. Applications that were previously contained in P memory only at startup, are now spread over several memory spaces.

There are two ways to attack this problem:

1. Force the ".xconst" etcetera sections back to the copytable. This brings back the waste of P memory, but if that is not a problem it is possible you can do this by using the locator description file. You have to specify the read–only sections in the description file and change their attributes:

   ```
   section selection=-w attr=i;
   ```

   These lines must be placed right under every position in X and Y memory spaces where,

   ```
   section selection=-w
   ```

   is written in the `.dsc` file.

2. Use a bootstrap loader that can handle the different memory spaces. The `bootrom` example contains such a loader and tools to generate a file for it, that you have to extend to match your application. This is the more elegant solution that we especially recommend if you are low on memory.

11.6 CALLING THE LOCATOR VIA THE CONTROL PROGRAM

It is recommended to call the locator via the control program **cc563**. The control program translates certain options for the locator (e.g., **–clas** to **–f4**). Other options (such as **–M**) are passed directly to the locator. Typical, you can use the control program to get an `.abs` file directly from `.c`, `.src`, `.asm` or `.obj` files. The invocation:

```
cc563 –M –g startup.asm calc.asm –o calc.abs
```

builds an absolute demo file called `calc.abs` ready for running via the CrossView debugger.

11.7 LOCATOR OUTPUT

The locator produces an absolute file and, if requested, a map file and/or an error file. The output file is absolute and in Intel Hex format, Motorola S–record format, Motorola CLAS object format or in IEEE–695 format, depending on the usage of the **–f** option. The default output name is `a.hex`, `a.sre`, `a.cld` or `a.abs`, respectively. Note that the debug information is not supported in the Motorola CLAS object format.

The map file (**–M** option) always has the same basename as the output object file, with an extension `.map`. The map file shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on address and sorted on symbol.

The map file also contains a part showing the sections defined in each source file (with their address and size).

The error output file (**–err** option) has the same name as the object output file, but with extension `.elc`. Errors occurred before the **–err** option is evaluated are printed on stderr.

11.8 LOCATOR MESSAGES

There are four kinds of messages: fatal messages, error messages, warning messages and verbose messages. Fatal messages are generated if the locator is not able to continue with its task due to the severity of the error. In those situations, the exit code will be 2. Error messages will be reported if an error occurred, not fatal for the locator. However, the output of the locator is not usable. The exit code in case of one or more error messages will be 1. Warning messages are generated if the locator detects potential errors, but the locator is unable to judge those errors. The exit code will be 0 in this case, indicating a usable `.abs` file. Of course, if the locator reports no messages, the exit code is also 0.

Each locator message has a built–in warning level. With option **–w***x* it is possible to suppress messages with a warning level above *x*.

Verbose messages are generated only if the verbose option (**–v**) is on. They report the progress of the locate process.

Locator messages have the following layout:

```
DSP563xx/6xx locator vx.y rz              SN00000000–127 (c) year TASKING, Inc.
lc563 W112 (3) calc.out: Copy table not referenced, initial data is not copied
```

11.9 ADDRESS SPACE

Figures 11–2 and 11–3 show the different address space mappings of the DSP5600x.

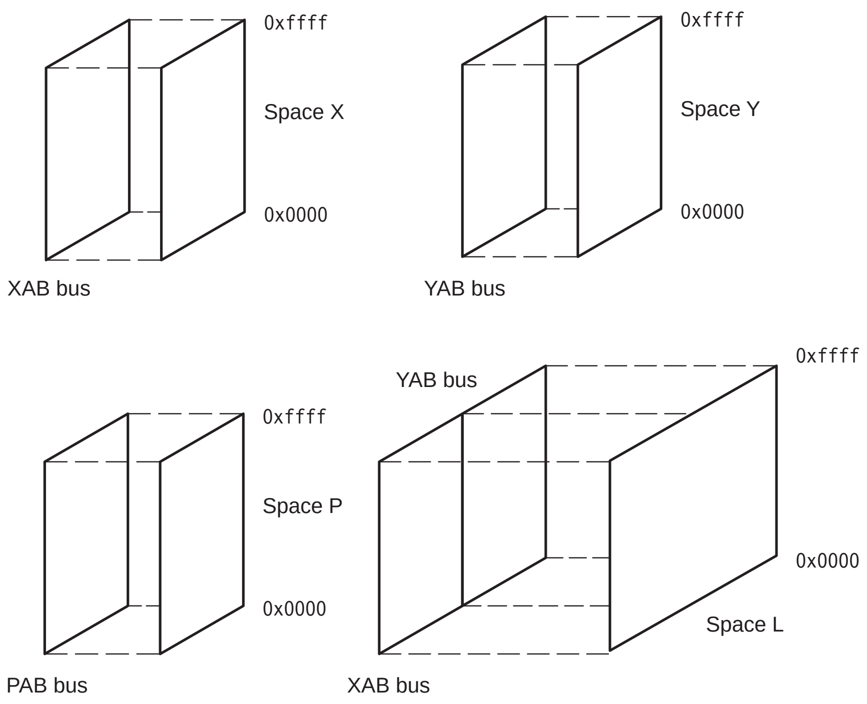


Figure 11–2: DSP5600x physical address space mapping

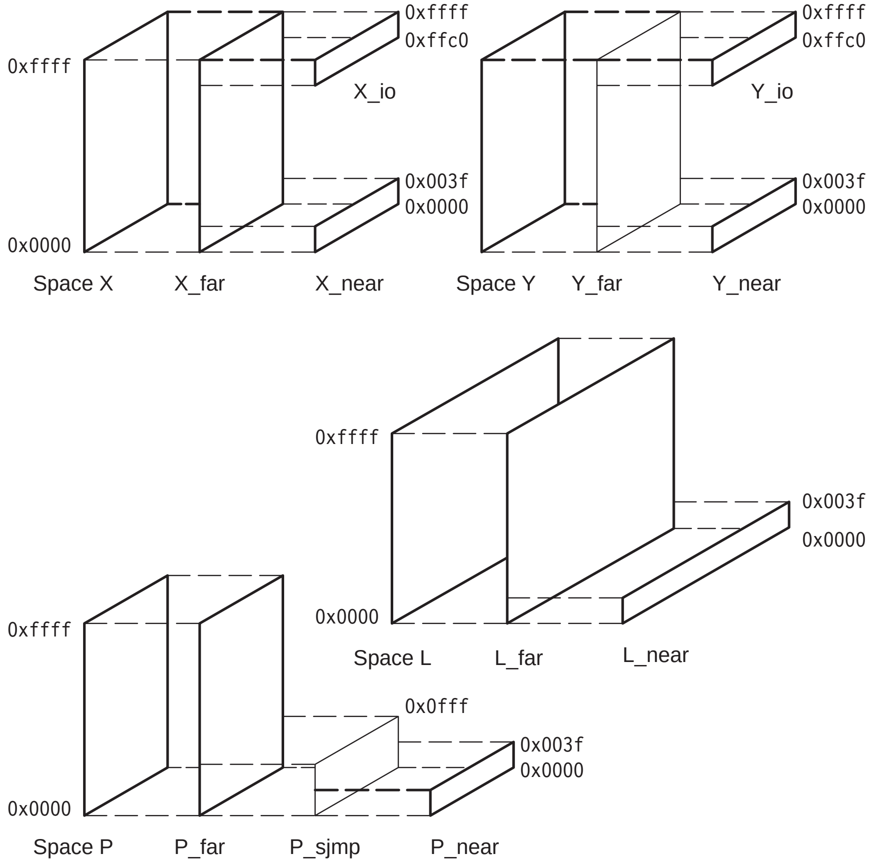


Figure 11–3: DSP5600x virtual address space mapping

11.10 LOCATOR LABELS

The locator assigns addresses to the following labels when they are referenced:

`F_lc_cp` :	Start of copy table The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the locator only if this label is used.
`F_lc_bs` :	Begin of stack space (using keyword `stack`).
`F_lc_es` :	End of stack space.
`F_lc_bh` :	Begin of heap space (using keyword `heap`).
`F_lc_eh` :	End of heap space.
`F_lc_b_`*`name`* :	Begin of section *name*.
`F_lc_e_`*`name`* :	End of section *name*.
`F_lc_cb_`*`name`* :	Begin of copy of section *name*.
`F_lc_ce_`*`name`* :	End of copy of section *name*.
`F_lc_u_`*`name`* :	User defined label. The label must be defined in the description file. For example: `label mylab;`
`F_lc_ub_`*`name`* :	Begin of user defined label. The label must be defined in the description file. For example: `label = mybuffer length=100;`
`F_lc_ue_`*`name`* :	End of user defined label.

At C level, all locator labels start with one underscore (the compiler adds an ’F’).

11.10.1 LOCATOR LABELS REFERENCE

This section contains a description of all locator labels. Locator labels are labels starting with **F_lc_**. They are ignored by the linker and resolved at locate time. Some of these labels are real labels at the beginning or the end of a section. Other labels have a second function, these labels are used to address locator generated data. The data is only generated if the label is used.

At C level, all locator labels start with one leading underscore (the compiler adds an 'F').

F_lc_b_section, F_lc_e_section

Syntax:

extern _X unsigned char **_lc_b_***section*[ ];
extern _X unsigned char **_lc_e_***section*[ ];

Description:

You can use the general locator labels **F_lc_b_***section* and **F_lc_e_***section* to obtain the addresses of section *section* in a program. The **b** version points to the start of the section, while the **e** version points to its end.

You can replace the dot before a section name by an underscore (_), making it possible to access these labels from ’C’. This convention introduces a possible name conflict. If, for example, both sections `.text` and `_text` exist, the general label `F_lc_b__text` is set to the start of `_text`. The label for section .text is only usable at assembly level with its real name. Of course, you should avoid such a conflict by not using section names with a leading underscore.

Example:

```
printf( ”Text size is 0x%x\n”,
        _lc_e__text – _lc_b__text );
```

F_lc_bh, F_lc_eh

Syntax:

extern _X unsigned char **_lc_bh**[ ];
extern _X unsigned char **_lc_eh**[ ];

Description:

All locator **h** labels are related to the heap. You can allocate a heap by defining it in a cluster description. See also the Delfee keyword **heap**.

F_lc_bh is a label at the begin of the heap. At 'C' level **_lc_bh** represent the heap. The label is defined as a char array, but an array of any basic type will do. **F_lc_eh** represents the end of the heap.

Example:

Heap definition:

```
block X_block {
     .
     .
     cluster X_clstr {
       amode X_far {
          heap length = 200;
          .
       }
     }
     .
}
```

Sbrk code:

```
extern _X unsigned char _lc_bh[ ];
extern _X unsigned char _lc_eh[ ];

static char *
sbrk( long length ) {
     .
     .

if ( (lastmem + length) > _lc_eh ) {
     return (char *) -1; /* overflow */
}
```

F_lc_bs, F_lc_es

Syntax:

extern _X unsigned char **_lc_bs**[ ];
extern _X unsigned char **_lc_es**[ ];

Description:

All locator **s** labels are related to the stack. You can allocate a stack by defining it in a cluster description. See also the Delfee keyword **stack**.

F_lc_bs is a label at the begin of the stack. At 'C' level **_lc_bs** represent the stack. The label is defined as a char array, but an array of any basic type will do. **F_lc_es** represents the end of the stack.

Example:

Stack definition:

```
block X_block {
     cluster X_clstr {
       amode X_far {
          stack length = 100;
          .
       }
     }
}
```

Stack initialization:

```
F_START:
     move #F_lc_bs, R7    ; set stack pointer to
                     ; begin of stack space
```

F_lc_cb_*section*, F_lc_ce_*section*

Syntax:

extern _X unsigned char **_lc_cb_***section*[ ];
extern _X unsigned char **_lc_ce_***section*[ ];

Description:

You can use the general locator labels **F_lc_cb_***section* and **F_lc_ce_***section* to obtain the addresses of the copy of section *section* in a program. The **b** version points to the start of the copy of the section, while the **e** version points to its end.

You can replace the dot before a section name by an underscore (_), making it possible to access these labels from 'C'. This convention introduces a possible name conflict. If, for example, both sections `.text_copy` and `_text_copy` exist, the general label `F_lc_cb__text` is set to the start of of `_text_copy`. The label for section `.text_copy` is only usable at assembly level with its real name. Of course, you should avoid such a conflict by not using section names with a leading underscore.

Example:

```
printf( ”Text size is 0x%x\n”,
    _lc_ce__text – _lc_cb__text );
```

F_lc_cp

Syntax:

extern char _X ***_lc_cp**;

Description:

The copy table is generated per process. Each entry in this table represents a copy or clearing action. Entries for the table are automatically generated by the locator for:

– All sections with attribute **b**, which must be cleared at startup time : a clearing action.

– All sections with attribute **i**, which must be copied from rom to ram at program startup: a copy action

F_lc_u_*identifier*

Syntax:

extern _X int **_lc_u_***identifier*[ ];

Description:

This locator label can be defined by the user by means of the Delfee keyword **label**. This label must be defined in the Delfee file without the prefix **F_lc_u_**. From assembly the label can be referenced with the prefix **F_lc_u_**, from C with the prefix **_lc_u_** (only the leading underscore).

Example:

In description file:

```
block X_block {
     cluster X_clstr {
       amode X_far {
          label = bstart;
          section text;
          label = bend;
       }
     }
     .
     .
     .
}
```

From C:

```
#include <stdio.h>
extern _X int _lc_u_bstart[];
extern _X int _lc_u_bend[];
int main()
{
  printf( ”Size of cluster X_clstr is %d\n”,
          (long)_lc_u_bend –
          (long)_lc_u_bstart );
}
```

From assembler:

```
extern F_lc_u_bstart
extern F_lc_u_bend
move #F_lc_u_bstart,r0
clra #F_lc_u_bend–#F_lc_u_bstart,b
rep  b
move a,x:(r0)+
```

F_lc_ub_*identifier*, F_lc_ue_*identifier*

Syntax:

extern _X int **_lc_ub_***identifier*[ ];
extern _X int **_lc_ue_***identifier*[ ];

Description:

These locator labels can be defined by the user by means of the Delfee keywords **reserved label=**. The locator labels specify the begin and end of a reserved area. The *identifier* is the name for the reserved area and must be defined in the Delfee file without the prefix **F_lc_ub_** or **F_lc_ue_**. From assembly the labels can be referenced with the prefix **F_lc_ub_** and **F_lc_ue_**, from C with the prefix **_lc_ub_** and **_lc_ue_** (only the leading underscore).

Example:

In description file:

```
block Y_block {
     cluster Y_clstr {
          attribute w;
          amode Y_far {
               section selection=w;
               reserved label=xvwbuffer length=0x10;
               // Start address of reserved area is
               // label F_lc_ub_xvwbuffer
               // End address of reserved area is
               // label F_lc_ue_xvwbuffer
          }
     }
}
```

From C:

```
#include <stdio.h>
extern _X int _lc_ub_xvwbuffer[];
extern _X int _lc_ue_xvwbuffer[];
int main()
{
  printf( ”Size of reserved area xvwbuffer is %d\n”,
          (long)_lc_ue_xvwbuffer –
          (long)_lc_ub_xvwbuffer );
}
```

From assembler:

```
     extern    F_lc_ub_xvwbuffer
     extern    F_lc_ue_xvwbuffer
     move #F_lc_ub_xvwbuffer,r0
     clra #F_lc_ue_xvwbuffer–#F_lc_ub_xvwbuffer,b
     rep  b
     move a,x:(r0)+
```

CHAPTER 12

UTILITIES

TASKING

CHAPTER 12

12.1 OVERVIEW

The following utilities are supplied with the Cross–Assembler for the DSP56xxx processor family which can be useful at various stages during program development.

ar563	An IEEE archiver. This is a librarian facility, which can be used to create and maintain object libraries.
cc563	A control program for the DSP563xx/DSP566xx toolchain.
mk563	A utility program to maintain, update, and reconstruct groups of programs.
pr563	An IEEE object reader that views the contents of files which have been created by a tool from the TASKING DSP563xx/DSP566xx toolchain.
byte_sel **order**	You can use these two special tools to create a bootable ROM image with the TASKING tools, in Intel–hex or Motorola S–record format. These tools extract the necessary data directly from the IEEE–695 (`.abs`) file, to make sure that the image in ROM exactly matches the program that was debugged.

For the DSP563xx/DSP566xx derivatives the executable names of the utilities are as mentioned above. For the DSP5600x derivatives the following executable names are used:

ar56	DSP5600x specific version of IEEE archiver.
cc56	DSP5600x specific version of control program.
mk56	DSP5600x specific version of make utility.
pr56	DSP5600x specific version of IEEE object reader.

When you use a **UNIX** shell (Bourne shell, C–shell), arguments containing special characters (such as '( )' and '?') must be enclosed with **" "** or escaped. The **-?** option (in the C–shell) becomes: **"-?"** or **-\?**.

The utilities are explained on the following pages.

12.2 AR563

Name

ar563	IEEE archiver and library maintainer (DSP563xx/6xx)
ar56	(DSP5600x)

Syntax

ar563 *key_option* [*option*]... *library* [*object_file*]...
ar563 –V
ar563 –? (UNIX C–shell : **"–?"** or **–\?**)

Description

With **ar563** you can combine separate object modules in a library file. The linker optionally includes modules from a library when a specific module resolves an external symbol definition in one of the modules that has been read before. The library maintainer **ar563** is a program to build library files and it offers the possibility to replace, extract or remove modules from an existing library.

key_option	one of the main options indicating the action **ar563** has to take. Key options may appear in any order, at any place.
option	optional sub–options as explained on the next pages.
library	is the library file.
object_file	is an object module to be added, extracted, replaced or removed from the library.

Options

You may specify options with or without a leading '**–**'. Options may occur in random order. You may also combine options. So **–xv** is allowed. **–V** and **–?** however, must be the only option on the command line.

Key options:

–d	Delete the named object modules from the library.
–m	Move the named object modules to the end of the library, or to another position as specified by one of the positioning options.

–p	Print the named object modules in the library on standard output.
–r	Replace the named object modules in the library if they exist. If they are not in the library, add them. If no names are given, only those object modules are replaced for which a file with the same name is found in the current directory. New modules are placed at the end.
–t	Print a table of contents of the library. If no names are given, all object modules in the library are printed. If names are given, only those object modules are tabled.
–x	Extract the named object modules from the library. If no names are given, all modules are extracted from the library. In neither case does **x** alter the library.

Other options:

–?	Display an explanation of options at `stdout`.
–V	Display version information at `stderr`.
–a *posname*	Append or move new object modules after existing module *posname*. This option can only be used in combination with the **m** or **r** option.
–b *posname*	Insert or move new object modules before existing module *posname*. This option can only be used in combination with the **m** or **r** option.
–c	Create the library file without notification if the library does not exist.
–f *file*	Read options from file *file*. '–' means `stdin`.
–o	Reset the last–modified date to the date recorded in the library. It can only be used in combination with the **x** option.
–s	Print a list of symbols. This option must be combined with **–t**.
–s1	Print a list of symbols. Each symbol is preceded by the library name and the name of the object file. This option must be combined with **–t**.

–u	Replace only those object modules with the last–modified date later than the library file. It can only be used in combination with the **r** option.
–v	Verbose. Under the verbose option, **ar563** gives a module–by–module description of the making of a new library file from the old library and the constituent modules. It can only be used in combination with the **d**, **m**, **r**, or **x** option.
–w*n*	Set warning level *n*.

Examples

1. Create library **clib.a** consisting of the modules **startup.obj**, and **calc.obj** :

```
ar563 cr clib.a startup.obj calc.obj
```

2. Extract all modules form library **clib.a** :

```
ar563 x clib.a
```

3. Print a list of symbols from library **clib.a** :

```
ar563 ts clib.a

startup.obj
   symbols:
        F_START
        F_copytable
        cptable_copy
        cptable_clr
calc.obj
   symbols:
        entry
```

4. Print a list of symbols from library **clib.a** in a different form:

```
ar563 ts1 clib.a

clib.a:startup.obj:F_START
clib.a:startup.obj:F_copytable
clib.a:startup.obj:cptable_copy
clib.a:startup.obj:cptable_clr
clib.a:calc.obj:entry
```

5. Delete module **calc.obj** from library **clib.lib** :

```
ar563 d clib.a calc.obj
```

12.3 BYTE_SEL

Name

byte_sel Create bootable ROM image in Intel Hex or Motorola S–record format.

Syntax

byte_sel [*option*]... *input_file*
byte_sel –h

Description

byte_sel processes the sequence of words produced by (a concatenation of the output of) the **order** utility and generates a Motorola S–record or an Intel–hex file.

Options

–a*hex_address*
Start address of output stream, default 0.

–b*N* Select byte to be output, default 1 (is least significant). This selects the *N*–th byte of the stream generated by **order**. For wordsize = 3 bytes, **–b3** gets the most significant byte and **–b1** gets the least significant byte.

–d*N* Drop the first *N* input lines, default 0.

–h Print usage and exit.

–i Output in Intel hex format.

–m Output in Motorola S–records (default).

Example

Create an absolute load file in Mototola S–record format:

```
order calc.abs > calc.tmp
byte_sel calc.tmp > calc.sre
```

12.4 CC563

Name

cc563 control program for the DSP563xx/DSP6xx toolchain

Syntax

cc563 [[*option*]... [*control*] ... [*file*]...]...
cc563 –V
cc563 –? (UNIX C–shell : **"–?"** or **–\?**)

Description

The control program **cc563** facilitates the invocation of the various components of the DSP563xx/6xx family toolchain from a single command line. **cc56** is the control program for the DSP5600x. The control program accepts source files and options on the command line in random order.

Options are preceded by a '–' (minus sign). The input *file* can have one of the extensions explained below.

The control program recognizes the following argument types:

- Arguments starting with a '–' character are options. Some options are interpreted by the control program itself; the remaining options are passed to those programs in the toolchain that accept the option.
- Arguments with a `.cc`, `.cxx` or `.cpp` suffix are interpreted as C++ source programs and are passed to the C++ compiler.
- Arguments with a `.c` suffix are interpreted as C source programs and are passed to the compiler.
- Arguments with a `.asm` suffix are interpreted as hand–written assembly source files which have to be passed to the assembler.
- Arguments with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Arguments with a `.a` suffix are interpreted as library files and are passed to the linker.
- Arguments with a `.obj` suffix are interpreted as object files and are passed to the linker.
- Arguments with a `.cln` suffix are interpreted as COFF object files and are passed to the linker.
- Arguments with a `.clb` suffix are interpreted as COFF library files and are passed to the linker.

- Arguments with a `.out` suffix are interpreted as linked object files and are passed to the locator. The locator accepts only one `.out` file in the invocation.
- An argument with a `.dsc` suffix is interpreted as a locator description file and is passed to the linker and the locator.

Normally, a control program tries to compile and assemble all source files to object files, followed by a link and locate phase which produces an absolute output file. There are however, options to suppress the assembler, linker or locator stage. The control program produces unique filenames for intermediate steps in the compilation process, which are removed afterwards. If the compiler and assembler are called subsequently, the control program prevents preprocessing of the compiler generated assembly file. Normally, assembly input files are preprocessed first.

Options

–? Display a short explanation of options at `stdout`.

–M*model* Specify the memory model to be used:

mixed	(**m**)	**cc56** only
reentrant	(**r**)	**cc56** only
static	(**s**)	**cc56** only
16–bit	(**16**)	**cc563** only
16/24–bit	(**1624**)	**cc563** only
24–bit	(**24**)	**cc563** only
DSP566xx	(**6**)	**cc563** only

This option is not supported for the **cc56** and **cc563**, they pass this option to the compiler and/or assembler and use it to select the C library of the selected model for the linker.

–S With this option the control program generates a Motorola compatible assembly file with COFF debug information.

–T *name* With this option the control program selects the target hardware *name* for the program. Name is the basename for both the startup file in the C library (*name*.`asm`), and the locator description file (*name*`.dsc`). Descriptive names are used for the preinstalled locator description files that can be found in the product `etc` directory.

–V The copyright header containing the version number is displayed, after which the control program terminates.

–Wa*arg* **–Wc***arg* **–Wcp***arg* **–Wlk***arg* **–Wlc***arg* **–Wpl***arg*	With these options you can pass a command line argument directly to the assembler (**–Wa**), C compiler (**–Wc**), C++ compiler (**–Wcp**), C++ pre–linker (**–Wpl**), linker (**–Wlk**) or locator (**–Wlc**). These options may be used to pass some options that are not recognized by the control program, to the appropriate program. The argument may be either directly appended to the option, or follow the option as a separate argument of the control program.
–c++	Specify that files with the extension `.c` are considered to be C++ files instead of C files. So, the C++ compiler is called prior to the C compiler. This option also forces the linker to link C++ libraries.
–c **–cc** **–cl** **–cs**	Normally, the control program invokes all stages to build an absolute file from the given input files. With these options it is possible to skip the C compiler, assembler, linker or locator stage. With the **–cc** option the control program stops after compilation of the C++ files and retains the resulting `.c` files. With the **–cs** option the control program stops after the compilation of the C source files (`.c`) and after preprocessing the assembly source files (`.asm`), and retains the resulting `.src` files. With the **–c** option the control program stops after the assembler, with as output one or more object files (`.obj`). With the **–cl** option the control program stops after the link stage, with as output a linker object file (`.out`).
–f *file*	Read command line arguments from *file*. The filename ”–” may be used to denote standard input. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility. Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.

2. To include whitespace in the argument, surround the argument with either single or double quotes.

3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:

   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.

   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

   Example:

   ```
   ”This has a single quote ’ embedded”
   ```

   or

   ```
   ’This has a double quote ” embedded’
   ```

   or

   ```
   ’This has a double quote ” and \
   a single quote ’”’ embedded”
   ```

4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non–quoted arguments, all whitespace on the next line will be stripped.

   Example:

   ```
   ”This is a continuation \
   line”
        –> ”This is a continuation line”
   ```

```
control(file1(mode,type),\
     file2(type))
     ->
control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

–clas
–ieee
–ihex
–srec
–tiof

With these options you can specify the locator output format of the absolute file. The output file can be a CLAS compatible file (`.cld`), IEEE–695 file (`.abs`), Intel Hex file (`.hex`), Motorola S–record file (`.sre`) or TIOF–695 file (`.abs`). The default output is IEEE–695 (`.abs`).

–nolib

With this option the control program does not supply the standard libraries to the linker. Normally the control program supplies the C, floating point and run–time libraries to the linker. Which libraries are needed is derived from the compiler options.

–o *file*

Normally, this option is passed to the locator to specify the output file name. When you use the **–cl** option to suppress the locating phase, the **–o** option is passed to the linker. When you use the **–c** option to suppress the linking phase, the **–o** option is passed to the assembler, provided that only one source file is specified. When you use the **–cs** option to suppress the assembly phase, the **–o** option is passed to the compiler. The argument may be either directly appended to the option, or follow the option as a separate argument of the control program.

–tmp

With this option the control program creates intermediate files in the current directory. They are not removed automatically. Normally, the control program generates temporary files for intermediate translation results, such as compiler generated assembly files, object files and the linker output file. If the next phase in the translation process completes successfully, these intermediate files will be removed.

–v	When you use the **–v** option, the invocations of the individual programs are displayed on standard output, preceded by a '+' character.
–v0	This option has the same effect as the **–v** option, with the exception that only the invocations are displayed, but the programs are not started.
–wc++	Enable C and assembler warnings for C++ files. The assembler and C compiler may generate warnings on C output of the C++ compiler. By default these warnings are suppressed.

Environment Variables used by cc56 / cc563

The control program uses the following environment variables:

TMPDIR	This variable may be used to specify a directory, which the control programs should use to create temporary files. When this environment variable is not set, temporary files are created in the directory "/tmp" on UNIX systems, and in the current directory on other operating systems.
CC56OPT	This environment variable may be used to pass extra options and/or arguments to each invocation of the control programs. The control programs process the arguments from this variable before the command line arguments.
CC563OPT	Same as CC56OPT, but now for **cc563**.
CC56BIN	When this variable is set, the control programs prepend the directory specified by this variable to the names of the tools invoked.
CC563BIN	Same as CC56BIN, but now for **cc563**.

12.5 MK563

Name

mk563 maintain, update, and reconstruct groups of programs

Syntax

mk563 [*option*]... [*target*]... [*macro***=***value*]...
mk563 –V
mk563 –? (UNIX C–shell: **"–?"** or **–\?**)

Description

mk563 takes a file of dependencies (a 'makefile') and decides what commands have to be executed to bring the files up–to–date. These commands are either executed directly from **mk563** or written to the standard output without executing them.

If no target is specified on the command line, **mk563** uses the first target defined in the first makefile.

Long filenames are supported when they are surrounded by double quotes ("). It is also allowed to use spaces in directory names and file names.

Options

–?	Show invocation syntax.
–D	Display the text of the makefiles as read in.
–DD	Display the text of the makefiles and 'mk563.mk'.
–G *dirname*	Change to the directory specified with *dirname* before reading a makefile. This makes it possible to build an application in another directory than the current working directory.
–K	Do not remove temporary files.
–S	Undo the effect of the **–k** option. Stop processing when a non–zero exit status is returned by a command.
–V	Display version information at stderr.

–W *target*	Execute as if this target has a modification time of ”right now”. This is the ”What If” option.
–a	Always rebuild a target without checking if it is out of date.
–d	Display the reasons why **mk563** chooses to rebuild a target. All dependencies which are newer are displayed.
–dd	Display the dependency checks in more detail. Dependencies which are older are displayed as well as newer.
–e	Let environment variables override macro definitions from makefiles. Normally, makefile macros override environment variables. Command line macro definitions always override both environment variables and makefile macros definitions.
–f *file*	Use the specified file instead of ’makefile’. A **–** as the makefile argument denotes the standard input.
–i	Ignore error codes returned by commands. This is equivalent to the special target .IGNORE:.
–k	When a nonzero error status is returned by a command, abandon work on the current target, but continue with other branches that do not depend on this target.
–m *file*	Read command line information from *file*. If *file* is a ’**–**’, the information is read from standard input.
–n	Perform a dry run. Print commands, but do not execute them. Even lines beginning with an @ are printed. However, if a command line is an invocation of **mk563**, that line is always executed.
–q	Question mode. **mk563** returns a zero or non–zero status code, depending on whether or not the target file is up to date.
–r	Do not read in the default file ’mk563.mk’.
–s	Silent mode. Do not print command lines before executing them. This is equivalent to the special target .SILENT:.
–t	Touch the target files, bringing them up to date, rather than performing the rules to reconstruct them.

–w Redirect warnings and errors to standard output. Without, **mk563** and the commands it executes use standard error for this purpose.

*macro***=***value*
Macro definition. This definition remains fixed for the **mk563** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate **mk563**'s but act as an environment variable for these. That is, depending on the **–e** setting, it may be overridden by a makefile definition.

Usage

Makefiles

The first makefile read is 'mk563.mk', which is looked for at the following places (in this order):

– in the current working directory
– in the directory pointed to by the HOME environment variable
– in the `etc` directory relative to the directory where **mk563** is located

Example (PC):

when **mk563** is installed in `\C563\BIN` the directory `\C563\ETC` is searched for makefiles.

Example (UNIX):

when **mk563** is installed in `/usr/local/c563/bin` the directory `/usr/local/c563/etc` is searched for makefiles.

It typically contains predefined macros and implicit rules.

The default name of the makefile is 'makefile' in the current directory. If this file is not found on a UNIX system, the file 'Makefile' is then used as the default. Alternate makefiles can be specified using one or more **–f** options on the command line. Multiple **–f** options act as if all the makefiles were concatenated in a left–to–right order.

The makefile(s) may contain a mixture of comment lines, macro definitions, include lines, and target lines. Lines may be continued across input lines by escaping the NEWLINE with a backslash (\). If a line must end with a backslash then an empty macro should be appended. Anything after a "#" is considered to be a comment, and is stripped from the line, including spaces immediately before the "#". If the "#" is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

An *include* line is used to include the text of another makefile. It consists of the word "include" left justified, followed by spaces, and followed by the name of the file that is to be included at this line. Macros in the name of the included file are expanded before the file is included. Include files may be nested.

An *export* line is used for exporting a macro definition to the environment of any command executed by **mk563**. Such a line starts with the word "export", followed by one or more spaces and the name of the macro to be exported. Macros are exported at the moment an export line is read. This implies that references to forward macro definitions are equivalent to undefined macros.

Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

`ifdef` *macroname*
if–lines
`else`
else–lines
`endif`

The *if–lines* and *else–lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else–lines* following it.

First the *macroname* after the `if` command is checked for definition. If the macro is defined then the *if–lines* are interpreted and the *else–lines* are discarded (if present). Otherwise the *if–lines* are discarded; and if there is an `else` line, the *else–lines* are interpreted; but if there is no `else` line, then no lines are interpreted.

When using the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

Macros

Macros have the form ‘WORD = text and more text’. The WORD need not be uppercase, but this is an accepted standard. Spaces around the equal sign are not significant. Later lines which contain $(WORD) or ${WORD} will have this replaced by ‘text and more text’. If the macro name is a single character, the parentheses are optional. Note that the expansion is done recursively, so the body of a macro may contain other macro invocations. The right side of a macro definition is expanded when the macro is actually used, not at the point of definition.

Example:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

‘$(FOOD)’ becomes ‘meat and/or vegetables and water’ and the environment variable FOOD is set accordingly by the export line. However, when a macro definition contains a direct reference to the macro being defined then those instances are expanded at the point of definition. This is the only case when the right side of a macro definition is (partially) expanded. For example, the line

```
DRINK = $(DRINK) or beer
```

after the export line affects ‘$(FOOD)’ just as the line

```
DRINK = water or beer
```

would do. However, the environment variable FOOD will only be updated when it is exported again.

You are advised not to use the double quotes (”) for long filename support in macros, otherwise this might result in a concatination of two macros with double quotes (”) in between.

Special Macros

MAKE	This normally has the value **mk563**. Any line which invokes MAKE temporarily overrides the **–n** option, just for the duration of the one line. This allows nested invocations of MAKE to be tested with the **–n** option.

MAKEFLAGS

This macro has the set of options provided to **mk563** as its value. If this is set as an environment variable, the set of options is processed before any command line options. This macro may be explicitly passed to nested **mk563**'s, but it is also available to these invocations as an environment variable. The **–f** and **–d** flags are not recorded in this macro.

PRODDIR

This macro expands the name of the directory where **mk563** is installed without the last path component. The resulting directory name will be the root directory of the installed DSP56xxx package, unless **mk563** is installed somewhere else. This macro can be used to refer to files belonging to the product, for example a library source file.

Example:

```
DOPRINT = $(PRODDIR)/lib/src/_doprint.c
```

When **mk563** is installed in the directory `/c563/bin` this line expands to:

```
DOPRINT = /c563/lib/src/_doprint.c
```

SHELLCMD

This contains the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it.

TMP_CCPROG

This macro contains the name of the control program. If this macro and the TMP_CCOPT macro are set and the command line argument list for the control program exceeds 127 characters then **mk563** will create a temporary file filled with the command line arguments. **mk563** will call the control program with the temporary file as command input file. This macro is only known by the DOS version of **mk563**.

TMP_CCOPT

This macro contains the option for the control program which tells the control program to read a file as command arguments. This macro is only known by the DOS version of **mk563**.

Example:

```
TMP_CCPROG= cc563
TMP_CCOPT = -f
```

$	This macro translates to a dollar sign. Thus you can use ”$$” in the makefile to represent a single ”$”.

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

$*	The basename of the current target.
$<	The name of the current dependency file.
$@	The name of the current target.
$?	The names of dependents which are younger than the target.
$!	The names of all dependents.

The $< and $* macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with F to specify the Filename components (e.g. ${*F}, ${@F}). Likewise, the macros $*, $< and $@ may be suffixed by D to specify the directory component.

The result of the $* macro is always without double quotes (”), regardless of the original target having double quotes (”) around it or not.
The result of using the suffix F (Filename component) or D (Directory component) is also always without double quotes (”), regardless of the original contents having double quotes (”) around it or not.

Functions

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. ’$(match arg1 arg2 arg3)’. All functions are built–in and currently there are five of them: `match`, `separate`, `protect`, `exist` and `nexist`.

`match`	The `match` function yields all arguments which match a certain suffix:

```
$(match .obj prog.obj sub.obj mylib.a)
```

will yield

```
prog.obj sub.obj
```

`separate` The `separate` function concatenates its arguments using the first argument as the separator. If the first argument is enclosed in double quotes then '\n' is interpreted as a newline character, '\t' is interpreted as a tab, '\ooo' is interpreted as an octal value (where, *ooo* is one to three octal digits), and spaces are taken literally. For example:

```
$(separate ”\n” prog.obj sub.obj)
```

will result in

```
prog.obj
sub.obj
```

Function arguments may be macros or functions themselves. So,

```
$(separate ”\n” $(match .obj $!))
```

will yield all object files the current target depends on, separated by a newline string.

`protect` The `protect` function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash.

Example:

```
echo $(protect I’ll show you the ”protect”
function)
```

will yield

```
echo ”I’ll show you the \”protect\”
function”
```

`exist` The `exist` function expands to its second argument if the first argument is an existing file or directory.

Example:

```
$(exist test.c cc563 test.c)
```

When the file `test.c` exists it will yield:

```
cc563 test.c
```

When the file `test.c` does not exist nothing is expanded.

`nexist` The `nexist` function is the opposite of the exist function. It expands to its second argument if the first argument is not an existing file or directory.

Example:

```
$(nexist test.src cc563 test.c)
```

Targets

A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
     [rule]
     ...
```

Any line which does not have leading white space (other than macro definitions) is a 'target' line. Target lines consist of one or more filenames (or macros which expand into same) called targets, followed by a colon (:). The ':' is followed by a list of dependent files. The dependency list may be terminated with a semicolon (;) which may be followed by a rule or shell command.

Special allowance is made on MS–DOS for the colons which are needed to specify files on other drives, so for example, the following will work as intended:

```
c:foo.obj : a:foo.c
```

If a target is named in more than one target line, the dependencies are added to form the target's complete dependency list.

The dependents are the ones from which a target is constructed. They in turn may be targets of other dependents. In general, for a particular target file, each of its dependent files is 'made', to make sure that each is up to date with respect to it's dependents.

The modification time of the target is compared to the modification times of each dependent file. If the target is older, one or more of the dependents have changed, so the target must be constructed. Of course, this checking is done recursively, so that all dependents of dependents of dependents of ... are up–to–date.

To reconstruct a target, **mk563** expands macros and functions, strips off initial white space, and either executes the rules directly, or passes each to a shell or COMMAND.COM for execution.

For target lines, macros and functions are expanded on input. All other lines have expansion delayed until absolutely required (i.e. macros and functions in rules are dynamic).

Special Targets

.DEFAULT:	The rule for this target is used to process a target when there is no other entry for it, and no implicit rule for building it. **mk563** ignores all dependencies for this target.
.DONE:	This target and its dependencies are processed after all other targets are built.
.IGNORE:	Non–zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying **–i** on the command line.
.INIT:	This target and its dependencies are processed before any other targets are processed.
.SILENT:	Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying **–s** on the command line.
.SUFFIXES:	The suffixes list for selecting implicit rules. Specifying this target with dependents adds these to the end of the suffixes list. Specifying it with no dependents clears the list.
.PRECIOUS:	Dependency files mentioned for this target are not removed. Normally, **mk563** removes a target file if a command in its construction rule returned an error or when target construction is interrupted.

Rules

A line in a makefile that starts with a TAB or SPACE is a shell line or rule. This line is associated with the most recently preceding dependency line. A sequence of these may be associated with a single dependency line. When a target is out of date with respect to a dependent, the sequence of commands is executed. Shell lines may have any combination of the following characters to the left of the command:

@ will not echo the command line, except if **–n** is used.

– **mk563** will ignore the exit code of the command, i.e. the ERRORLEVEL of MS–DOS. Without this, **mk563** terminates when a non–zero exit code is returned.

+ **mk563** will use a shell or COMMAND.COM to execute the command.

If the '+' is not attached to a shell line, but the command is a DOS command or if redirection is used (<, |, >), the shell line is passed to COMMAND.COM anyway. For UNIX, redirection, backquote (`) parentheses and variables force the use of a shell.

You can force **mk563** to execute multiple command lines in one shell environment. This is accomplished with the token combination ';\'.

Example:

```
cd c:\c563\bin ;\
cc563 –V
```

The ';' must always directly be followed by the '\' token. Whitespace is not removed when it is at the end of the previous command line or when it is in front of the next command line. The use of the ';' as an operator for a command (like a semicolon ';' separated list with each item on one line) and the '\' as a layout tool is not supported, unless they are separated with whitespace.

mk563 can generate inline temporary files. If a line contains '<<WORD' then all subsequent lines up to a line starting with WORD, are placed in a temporary file. Next, '<<WORD' is replaced by the name of the temporary file.

No whitespace is allowed between '<<' and 'WORD'.

Example:

```
lk563 –o $@ –f <<EOF
     $(separate ”\n” $(match .obj $!))
     $(separate ”\n” $(match .a $!))
     $(LDFLAGS)
EOF
```

The three lines between the tags (EOF) are written to a temporary file (e.g. ”\tmp\mk2”), and the command line is rewritten as ”lk563 –o $@ –f \tmp\mk2”.

Implicit Rules

Implicit rules are intimately tied to the .SUFFIXES: special target. Each entry in the .SUFFIXES: list defines an extension to a filename which may be used to build another file. The implicit rules then define how to actually build one file from another. These files are related, in that they must share a common basename, but have different extensions.

If a file that is being made does not have an explicit target line, an implicit rule is looked for. Each entry in the .SUFFIXES: list is combined with the extension of the target, to get the name of an implicit target. If this target exists, it gives the rules used to transform a file with the dependent extension to the target file. Any dependents of the implicit target are ignored.

If a file that is being made has an explicit target, but no rules, a similar search is made for implicit rules. Each entry in the .SUFFIXES: list is combined with the extension of the target, to get the name of an implicit target. If such a target exists, then the list of dependents is searched for a file with the correct extension, and the implicit rules are invoked to create the target.

Examples

This makefile says that `prog.out` depends on two files `prog.obj` and `sub.obj`, and that they in turn depend on their corresponding source files (`prog.c` and `sub.c`) along with the common file `inc.h`.

```
LIB  =     -lc24

prog.out:  prog.obj sub.obj
     lk563 prog.obj sub.obj $(LIB) -o prog.out

prog.obj:  prog.c inc.h
     c563  prog.c
     as563 prog.src

sub.obj:   sub.c inc.h
     c563  sub.c
     as563 sub.src
```

The following makefile uses implicit rules (from `mk563.mk`) to perform the same job.

```
LDFLAGS    =    -ls
prog.out:  prog.obj sub.obj
prog.obj:  prog.c inc.h
sub.obj:   sub.c inc.h
```

Files

makefile	Description of dependencies and rules.
Makefile	Alternative to makefile, for UNIX.
mk563.mk	Default dependencies and rules.

Diagnostics

mk563 returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

12.6 ORDER

Name

order Order the contents of a TASKING IEEE–695 file.

Syntax

order [*option*]... *input_file*
order –h

Description

order reads a TASKING IEEE–695 file, orders its contents by address and generates a sequence (consisting of a size word, the range start address and the range contents) for each contiguous range found in the input.

The output of the **order** command can be processed by **byte_sel**.

Options

–B	Byte wide (big endian order), default MAU wide.
–b	Byte wide (little endian order), default MAU wide. The input words will be put out as the least significant bytes in the output stream, in little–endian order.
–h	Print usage and exit.
–m*N*	*N* bytes per MAU in output, default 3. This option sets the word size (in bytes). The current limit is 4 bytes. All calls to **order** must currently be presented the same number of bytes.
–s	Add an extra record containing the start address, preceded by a length record with value 0.
–t*cpu*	Select the target CPU, default 563xx. Possible values are 5600x, 563xx or 566xx.
–v	Verbose. Show the address ranges that are being put out.
–y*mem*	Select the memory space to be output, default all. Possible values are P, X or Y. The memory space is encoded in the highest one or two bits of the size word: 0 for P memory, 10 for X memory and 11 for Y memory.

–z Fill gaps with zeroes, default: don’t. Only one range will be generated. This can be expensive!

Output

```
{
    memory space | size of code/data range
    start address of code/data range
    image of code/data range (’size’ words)
} *
[0
start address of application]
```

12.7 PR563

Name

pr563	IEEE object reader (DSP563xx/DSP566xx) Displays the contents of a relocatable object file or an absolute file
pr56	IEEE object reader (DSP5600x)

Syntax

pr563 [*option*]... *file*
pr563 –V
pr563 –? (UNIX C–shell: **"–?"** or **–\?**)

Description

pr563 gives you a high level view of an object file which has been created by a tool from the TASKING DSP563xx/6xx toolchain. Note that **pr563** is not a disassembler. Use **pr56** with the DSP5600x toolchain.

Options

Options start with a '–' sign and can be combined after a single '–'. There are options to print a specific part of an object file. For example, with option **–h** you can display the header part, the environment part and the AD/extension part as a whole. These parts are small, and you cannot display these parts separately. If you do not specify a part, the default is **–hscegd0i0** (all parts, the debug part and the image part displayed as a table of contents).

Furthermore, there are some additional options by which you can control the output.

Input Control Option

–f *file* Read command line information from *file*. If *file* is a '**–**', the information is read from standard input.

Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.

More than one **–f** option is allowed.

Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

   Example:

   ```
       ”This has a single quote ’ embedded”
   ```

   or

   ```
       ’This has a double quote ” embedded’
   ```

   or

   ```
       ’This has a double quote ” and \
       a single quote ’”’ embedded”
   ```

4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non–quoted arguments, all whitespace on the next line will be stripped.

Example:

```
”This is a continuation \
line”
     -> ”This is a continuation line”

control(file1(mode,type),\
     file2(type))
     ->
control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

Output Control Options

–H or **–?**	Display an explanation of options at `stdout`.
–V	Display version information at `stderr`.
–W*n*	Set output width to *n* columns. Default 128, minimum 78.
–l*n*	Level control, see paragraph 12.7.3.
–o*file*	Name of the output file, default `stdout`.
–v	Print the selected parts in a verbose form.
–v*n*	Print level *n* verbose, see paragraph 12.7.3.
–w*n*	Suppress messages above warning level *n*.

Display Options

–c	Print call graphs.
–d	Print all debug info except for the global types.
–d0	Print table of contents for the debug part.
–d*n*	Print debug info from file number *n*.
–e	Print variables with external scope.
–e1	Print variables with external scope and precede symbol name with name of the object file.
–g	Print global types.
–h	Print general file info.

–i	Print all section images.
–i0	Print table of contents for the image part.
–i*n*	Print image of section *n*.
–s	Print section info.

12.7.1 PREPARING THE DEMO FILES

There are three files which are used in this chapter to show how you can use **pr563**. These files are:

calc.obj
calc.out
calc.abs

If you want to try the examples yourself, prepare these files by copying the calc example files to a working directory. Be sure that the DSP56xxx tools can be found via a search path. Make the files with the following command:

```
cc563 –M –g –tiof –nolib startup.asm calc.asm –o calc.abs
–tmp
```

12.7.2 DISPLAYING PARTS OF AN OBJECT FILE

12.7.2.1 OPTION -h, DISPLAY GENERAL FILE INFO

The **–h** option gives you general information of the file. The invocation:

```
pr56 –h calc.out
```

Gives the following information:

```
File name    = calc.out:
Format       = Relocatable
Produced by  = DSP5600x object linker
Date         = apr 11, 1999 12:12:54h
```

This output speaks for itself. You may combine the **–h** switch with the verbose option:

```
pr56 –hv calc.out
```

The output is extended with more general information of less importance:

```
File name    = calc.out:
Format       = Relocatable
Produced by  = DSP5600x object linker
Date         = apr 11, 1999 12:12:54h
Obj version  = 1.1
Processor    = 5600x
Address size = 24 bits
Byte order   = Least significant byte at lowest address
Host         = Sun

Part                    File offset   Length
----------------------------------------------------
Header part             0x00000000    0x00000054
AD Extension part       0x00000054    0x00000033
Environment part        0x00000087    0x0000002e
Section part            0x000000b5    0x00000057
External part           0x0000010c    0x000000d1
Debug/type part         0x000001d2    0x000000a2
Data part               0x00000274    0x00000427
Module end              0x0000069b
```

The table gives you the file offsets and the length of the main object parts.

12.7.2.2 OPTION -s, DISPLAY SECTION INFO

With the **–s** option, you can obtain the section information from an object module. The section **contents** can be obtained with the **–i** option, see 12.7.2.7.

```
pr563 –s calc.out

Section   Size
-----------------------
Nameless  0x02
.text     0x40
.xbss     0x01
.ptext    0x29
.xdata    0x02
```

Note that the section information is not available any more in a located file. Once located, the separate *sections* are combined to new *clusters*. For an absolute file ’**pr563 –s**’.will give the *cluster* information:

```
pr563 –s calc.abs

Section   Size
------------------
P_clstr   0x76
X_clstr   0x1003
```

The locate map shows you which section is located in which cluster. Of course, you can also use the verbose option to see all section information available:

```
pr563 –sv calc.out
```

```
Section  Size Addr Algn Page Mau Attributes
-----------------------------------------------------------------------------
Nameless 0x02 0x00 0x1  –    –   Execute ZeroPage Space 1 Abs Separate
.text    0x40 –    0x1  –    –   Execute ZeroPage Space 1 Cumulate
.xbss    0x01 –    0x1  –    –   Write Space 2 Cleared Cumulate
.ptext   0x29 –    0x1  –    –   Execute ZeroPage Space 1 Cumulate
.xdata   0x02 –    0x1  –    –   Write Space 2 Initialized Cumulate
```

The first two columns give you the section name and the section size. The column ’Address’ gives you the section address, or a ’–’ if the section is still relocatable. The section alignment is always 1 for the DSP56xxx. The page size is valid only for the short sections. MAU is the minimum addressable unit of an address space (in bits). There are two main groups of section attributes, the allocation attributes, used by the locator and the overlap attributes, used by the linker:

Allocation attributes	
Write	Must be located in ram
ReadOnly	May be located in rom
Execute	May be located in rom
Space *num*	Must be located in addressing mode *num*
Abs	Already located by the assembler
Cleared	Section must be initialized to '0'
Initialized	Section must be copied from ram to rom
Scratch	Section is not filled or cleared

Table 12–1: Allocation attributes

Overlap attributes	
MaxSize	Use largest length encountered
Unique	Only one section with this name allowed
Cumulate	Concatenate sections with the same name to one bigger section
Overlay	Sections with the name *name@func* must be combined to one section *name*, according to the rules for *func* obtained from the call graph.
Separate	Sections are not linked.

Table 12–2: Overlap attributes

12.7.2.3 OPTION -c, DISPLAY CALL GRAPHS

The call graph is used by the linker overlaying algorithm. Once a file is linked and overlaying is done, the call graph information is removed from the object file. If you try to see the call graph in `calc.out` you will get the message 'No call graph found'.

The file `calc.obj` is not yet linked. You can use this file to see what a call graph looks like:

```
pr563 –c calc.obj
```

Because the `calc` example does not contain any sections which need to be overlaid you will again get the message 'No call graph found'. The following is just an example of what a call graph could look like:

```
Call graph(s)
=============

Call graph 0:

main()
  ->See call graph 1
  ->See call graph 4
  ->See call graph 2
  _exit()
  print_str()
  clear_screen()


Call graph 1:

queens?find_legal_row()
  ->See call graph 1
  ->See call graph 2
  abs()
  ->See call graph 3
```

Each call graph consists of a function (`main` in graph 0), followed by a list of functions and/or other graphs, which are called by the first function. The functions and call graphs called by this function are indented by two spaces. If a function calls other functions, those functions are listed again with another indentation of two spaces.

As you can see, there are references from one call graph to another. Call graph 1 even calls itself!! This means that function `find_legal_row()` is a recursive function. If you use the verbose switch the output is somewhat nicer:

```
main()
  |
  +--->See call graph 1
  |
  +--->See call graph 4
  |
  +--->See call graph 2
  |
  +--exit()
  |
  +--print_str()
  |
  +--clear_screen()
```

The function `find_legal_row` from call graph 1 is a static function. In order to avoid name conflicts, the source name is added to this function name.

If you want a call graph with resolved call graph references, you can use the linker to generate one:

```
lk563 –o call.out –Mcr calc.obj
```

Option **–M** tells the linker to generate a `.lnl` file. This file contains the call graph in the verbose layout. Option **–c** causes the linker to generate a `.cal` file. This file contains also the (same) call graph, but in the compact (non verbose) layout. Option **–r** tells the linker that this is an incremental link.

12.7.2.4 OPTION -e, DISPLAY EXTERNAL PART

In the external part of an object file, you can find all symbols used at link time. These symbols have an external scope. With the **–e** option (or **–e0**) **pr563** displays the external symbols:

```
pr563 –e calc.out
```

```
Variable      S  Address/Size
-----------------------------
F_START       I  .text + 0x00
F_copytable   I  .text + 0x0c
cptable_copy  I  .text + 0x1d
cptable_clr   I  .text + 0x33
entry         I  .ptext + 0x1f
F_lc_bs       X  –
F_lc_b__xbss  X  –
F_lc_e__xbss  X  –
F_lc_cp       X  –
```

With option **–e1** also the name of the output object file is displayed.

```
pr563 –e1 calc.out
```

```
Variable               S  Address/Size
--------------------------------------
calc.out:F_START       I  .text + 0x00
calc.out:F_copytable   I  .text + 0x0c
calc.out:cptable_copy  I  .text + 0x1d
calc.out:cptable_clr   I  .text + 0x33
calc.out:entry         I  .ptext + 0x1f
calc.out:F_lc_bs       X  –
calc.out:F_lc_b__xbss  X  –
calc.out:F_lc_e__xbss  X  –
calc.out:F_lc_cp       X  –
```

The first column contains the name of the symbol. In general, this symbol is a high level symbol with an 'F' added at the front. The next column gives you the symbol status. This can be **I** for a defined symbol, and **X** for a symbol which is referred to, but which is not yet defined. In the last column you can find the symbols address. If this address is still relocatable, the section offsets are printed in the form '*section* + offset'. If a symbol has already received an absolute address, this address is printed. Symbols that are not yet defined (marked with a **X**) have a dash printed as address, indicating *unknown*.

You can add the verbose option as usual. With verbose on more information is printed:

```
pr563 –ev calc.out
```

```
Variable        S   Type  Attrib  MAU  Amod Address/Size
-------------------------------------------------------------
F_START         I   –     –       24      1 .text + 0x00
F_copytable     I   –     –       24      1 .text + 0x0c
cptable_copy    I   –     –       24      1 .text + 0x1d
cptable_clr     I   –     –       24      1 .text + 0x33
entry           I   –     –       24      1 .ptext + 0x1f
F_lc_bs         X   –     –       24      2 –
F_lc_b__xbss    X   –     –       24      0 –
F_lc_e__xbss    X   –     –       24      0 –
F_lc_cp         X   –     –       24      0 –
```

Four additional columns appear. The Type column gives you the symbol type, if available. You can find the meaning of the types in the global type part, section 12.7.2.5. The global types are used to type check the symbols during linking. The Attribute column specifies the attribute of the symbol, if available. For example, the attribute value 0x0020 indicates that the symbol is generated by the assembler. The MAU colomn indicates the minimum addressable unit in bits. So, MAU 24 means the symbol is 24–bit addressable. The Amod column lists the addressing mode of the symbol.

12.7.2.5 OPTION -g, DISPLAY GLOBAL TYPE INFORMATION

The linker uses the global type information to check on type mismatches of the symbols in the external part. This information is always available, unless you explicitly suppress the generation of these types with option **–gn** at compile time. Of course, type checking can only be done if the types are available. The global types in `calc.out`:

```
pr563 –g calc.out
```

In this example you will get the message ’No global types available’. The following is just an example of what the global type information could look like:

```
Tp#  Mnem Name Entry
---------------------------
101  X    –    0, T10, 0, 0
102  X    –    0, T1, 0, 0
103  X    –    0, T1, 0, 1, T104
104  P    –    T105
105  n    –    T2, 1
106  X    –    0, T1, 0, 1, T10
107  X    –    0, T10, 0, 1, T10
108  X    –    0, T1, 0, 2, T109, T109
109  T    Byte T3
10a  X    –    0, T1, 0, 1, T109
...
10f  X    –    0, T1, 0, 3, T12, T110, T12
110  O    –    T111
111  n    –    T2, 0
112  Z    –    T2, 13
113  Z    –    T2, 7
```

In the first column you find the type index. This is the number by which the type is referred to. This number is always a hexadecimal number. Numbering starts at 0x101, because the indices less than 0x100 are reserved for, so–called, ’basic types’. The second column contains the type mnemonic. This mnemonic defines the new ’high level’ type. In the Name column you will find the name for the type, if any.

The last column contains type parameters. They tell you which (basic) types a high level type is based on and give other parameters such as modes and sizes. Types are preceded by a **T**. So, in the example above, type 105 is based upon type 2 (T2 in the parameter list) and type 103 is based upon type 1 and type 104.

In the next table you can find an overview of the basic types:

Type index	Type	Meaning
1	void	–
2	char	8 bits signed
3	unsigned char	8 bits unsigned
4	short	16 bits signed
5	unsigned short	16 bits unsigned
6	long	32 bits signed
7	unsigned long	32 bits unsigned

Type index	Type	Meaning
10	float	32 bit floating point
11	double	64 bit floating point
16	int	16 bits signed
17	unsigned int	16 bits unsigned

Table 12–3: Basic types

Mnemonic	Description	Parameters
G	generalized structure	size, [member, T*index*, offset, size]...
N	enumerated type	[name, value]...
n	pointer qualifier	T*index*, memspace
O	small pointer	T*index*
P	large pointer	T*index*
Q	type qualifier	q–bits, T*index*
S	structure	size, [member, T*index*, offset]...
T	typedef	T*index*
t	compiler generated type	T*index*
U	union	size, [member, T*index*, offset]...
X	function	x–bits, T*index*, 0, nbr–arg, [T*index*]...
Z	array	T*index*, upper–bound
g	bit type	sign, nbr–of–bits

Table 12–4: Type mnemonics

The type mnemonics define the class of the newly created type. The next table shows the type mnemonics with a short description:

The T*index* for mnemonic n, O, P, Q, T, t and Z are the types upon which the new type is built. The T*index* for the union and the structures are the type indices for the members. For the function type, the first T*index* is the return type of the function. The second T*index* is repeated for each parameter, and gives the type of each parameter. The value –1 (0xffffffff) always means 'unknown'. This can occur with a function type if the number of parameters is unknown, or with an array if the upper bound in unknown. The sizes and offset for the generalized structure are in bits. The first size is the size of the structure, the second size is the size for the member.

The type information obtained with the **–g** switch has no verbose equivalent.

12.7.2.6 OPTION -d, DISPLAY DEBUG INFORMATION

The **–d** switch has two variants. With **–d0** you get a table of contents:

```
pr563 –d0 calc.out

Choose option –d with the number of the file:
 1 – startup.obj
 2 – calc.obj
```

Now, you can use **–d***n* to examine a single (linked) file. For instance, **–d2** shows you only the debug info of `calc.obj`. It is also possible to see all debug info, by using option **–d** without a value.

The **–d** switch without the verbose option **–v** shows you only local variables and procedure information. If you combine the **–d** switch with the verbose switch **–v**, also local type info, line numbers, stack update information and more procedure information is displayed.

In the example you are using the verbose switch. Where required, the remark 'Only with verbose on' will be given.

```
pr563 –d2v calc.out
```

The object reader starts with a header, followed by the local type information:

```
************************************
*   O b j e c t   c a l c . o b j   *
************************************

M o d u l e   i n f o
=====================

Type info calc.obj:
===================

No local types available
```

This type info is only printed if you use the verbose option **–v**. The information found in this table is exactly the same as the information explained for the global type information, see 12.7.2.5.

After the local types, you will find the local symbols.

```
Symbols calc.obj:
=================

Variable   S  Type  Attrib  MAU Amod Address/Size
-----------------------------------------------------
factorial  N  –     0x0020  24     1 –
compute    N  –     0x0020  24     1 –
endfunc    N  –     0x0020  24     1 –
val        N  –     0x0020  24     2 –
cll        N  –     0x0020  24     2 –
zero       N  –     0x0020  24     2 –
```

The value for the symbol status in the external part was an **I** or an **X**. Here, you can see a new letter. The **N** stands for a local symbol. Other possible entires can have the letter **G** or **S**. They are no symbols, but procedures. These procedures are printed at this place in order to define their relative position. The actual procedure information is given in the next block of information. Here you can find the additional procedure information. The procedure block is printed only if you use the verbose switch:

```
Procedures calc.obj:
====================

No procedures
```

The following is an example of some procedures:

```
Name              S     Additional information
-----------------------------------------
main              G     0x00, 0x00, T101, QUEENS_PR + 0x00,
                        ( QUEENS_PR + 0x49 ) – 0x01
find_legal_row    S     0x00, 0x00, T120, QUEENS_PR + 0x49,
                        ( QUEENS_PR + 0x156 ) – 0x01
display_board     S     0x00, 0x00, T10a, QUEENS_PR + 0x156,
                        ( QUEENS_PR + 0x2a4 ) – 0x01
display_field     S     0x00, 0x00, T121, QUEENS_PR + 0x2a4,
                        ( QUEENS_PR + 0x302 ) – 0x01
display_status    S     0x00, 0x00, T103, QUEENS_PR + 0x302,
                        ( QUEENS_PR + 0x31d ) – 0x01
```

The first two columns are the same as those in the local variable table. The **G** stands for an external (global) function, the **S** for a static (local) function.

Each function has 5 parameters with the following meaning:

param #1	Frame type, not used
param #2	Frame size, the distance from the stack pointer before the function call to the stack position just after the local variables.
param #3	The type of the function
param #4	The start address of the function. In a relocatable object the syntax '*section* + offset' is used.
param #5	The last function address. See also param #4.

Next in the debug info is the line number information and the stack information. Both items are only printed if you had turned the verbose switch on:

```
Lines include/stdarg.h:
=======================
No line info available

Lines include/stdio.h:
======================
No line info available
```

```
Lines queens.c:
===============

Address                | Line   Address                | Line   Address    ...
---------------------------     ---------------------------     ---------------
QUEENS_PR + 0x000000   | 52     QUEENS_PR + 0x0000c2   | 90     QUEENS_PR + ...
QUEENS_PR + 0x000000   | 53     QUEENS_PR + 0x0000d9   | 101    QUEENS_PR + ...
QUEENS_PR + 0x000006   | 55     QUEENS_PR + 0x0000d9   | 103    QUEENS_PR + ...
         .                               .                               .
         .                               .                               .
         .                               .                               .
QUEENS_PR + 0x0000bd   | 98     QUEENS_PR + 0x00018e   | 133    QUEENS_PR + ...
QUEENS_PR + 0x0000c0   | 99     QUEENS_PR + 0x000190   | 136    QUEENS_PR + ...
QUEENS_PR + 0x0000c2   | 100    QUEENS_PR + 0x00019f   | 137

Stack info include/stdarg.h:
============================
No stack info available

Stack info include/stdio.h:
===========================
No stack info available

Stack info queens.c:
====================
No stack info available
```

The stack info gives the actual stack position for each executable address. This value is measured from the start position, just after the functions local variables to the actual stack position. If you push one byte on stack, the delta will be increased by one.

The debug info per module ends with a block for each function. Within this block the local variables per function are displayed:

```
P r o c e d u r e   i n f o
===========================

Procedure find_legal_row:
=========================

Symbols find_legal_row:
=======================

Variable  S  Type    Attrib  Mau  Amod  Address/Size
----------------------------------------------------
accepted  N  0x0109  0x0004   0     0   QUEENS_DA +
0x09
row       N  0x0109  0x0805   0     0   0x02
col       N  0x0109  0x0805   0     0   0x03
chk_row   N  0x0109  0x0005   0     0   0x01
chk_col   N  0x0109  0x0005   0     0   0x00

E n d   o f   p r o c e d u r e   i n f o
=========================================
```

12.7.2.7 OPTION –i, DISPLAY THE SECTION IMAGES

As with the **–d** option, you can ask a table with available section images by specifying option **–i0**:

```
pr563 –i0 calc.out
```

```
Choose option –i with the number of the section:
 1 – Nameless
 2 – .text
 3 – .xbss
 4 – .ptext
 5 – .xdata
```

You can select the image to display by specifying the image number:

```
pr563 –i4 calc.out
```

```
Section .ptext:
===============

05 5f 3c 20 5f 1b 57 f4 00 00 00 02 20 00 0d 0a
f0 af rr rr rr 56 f4 00 00 00 01 0a f0 80 rr rr
rr 77 f4 00 ff ff ff 46 f4
```

It is also possible to get the section offsets or absolute addresses by specifying the verbose flag:

pr563 –i4v calc.out

```
Section .ptext:
===============

00      05 5f 3c 20 5f 1b 57 f4 00 00 00 02 20 00 0d 0a     ._< _.W..... ...
10      f0 af rr rr rr 56 f4 00 00 00 01 0a f0 80 rr rr     .....V..........
20      rr 77 f4 00 ff ff ff 46 f4                          .w.....F.
```

The dump always shows the hexadecimal byte value per address. Sometimes however, this is not possible. First of all, it is possible that a certain byte cannot be determined because it is not yet relocated. In this case the byte is represented as **rr**.

Secondly, it is possible that there is no section image allowed. This is for instance the case for sections that are cleared during startup. The section with index 5 (`.xbss`) is such a section. After the invocation (verbose on) the reader prints:

pr563 –i3v calc.out

```
Section .xbss:
==============

No image allowed, cleared during startup
```

It is possible that you read an absolute file. In the absolute file it is possible to combine different sections to new clusters. These clusters do not have the same attributes as the sections and the reader does no longer know where the overlay area is positioned:

pr563 –v –i2 calc.abs

```
Section X_clstr:
================

00      ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss     ................
10      ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss     ................
20      ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss     ................
```

As you see, the reader only prints bytes that it actually can read from the object file. The **ss** in the dump means *scratch* memory. It may or may not be initialized by the start–up code. This information is not available anymore to the reader. The start–up code can use a locator generated table to get the information. See the *Locator* chapter.

12.7.3 VIEWING AN OBJECT AT LOWER LEVEL

12.7.3.1 OBJECT LAYERS

As with the well known OSI layer model for communication, you can also distinguish layers in an object file. The object file is a medium for the compiler which lets the compiler communicate with the debugger or the target board. The lowest level can be classified as mass storage, mostly the disc. The lowest viewable level for the readers concern are the raw bytes.

pr563 knows this layer as *level 0*.

Of course, the bytes in level 0 have a meaning. Because the object format is an format according to IEEE 695, the object file is a collection of MUFOM commands. The general idea is, that an object producing tool sends commands to a object consuming tool. These commands are described in detail by the official IEEE standard[1]. The raw bytes from level 0 appear to be encoded MUFOM commands. The MUFOM commands are interpreted in a layer just above the raw bytes layer.

pr563 knows this layer as *level 1*.

The next layer is the MUFOM environment, the type and section tables are built, values are assigned, attributes are set just by performing the MUFOM commands. The IEEE document describes also some predefined meanings about scope, section attributes naming conventions for MUFOM variables. This knowledge is available in the highest MUFOM layer.

pr563 knows this layer as *level 2.*

[1] IEEE Trial Use Standard for Microprocessor Universal Format for Object Modules (IEEE std. 695), IEEE Technical Committee on Microcomputers and Microprocessors of the IEEE Computer Society, 1990.

With these first layers, the compiler and debugger/target board have a perfect communication channel. The next layers (not supported by the reader at this moment) define a protocol between compiler and debugger about target and language specific information.

In the next sections you can find some examples about the use of the reader at lower levels. Until now, you used the default level of the reader, level 2.

12.7.3.2 THE LEVEL OPTION -l*n*

Level 1

Switching to another level is simple. You can use the **–l** option with the level you want to see. As an example, the section part of `calc.out` at level 1:

```
pr563 –l1 –s calc.out
```

```
ST:  1, XAZSN,
AS:  L1, 0x0
AS:  S1, 0x2
ST:  2, XZCN, .text
AS:  S2, 0x40
ST:  3, WBY2CN, .xbss
AS:  S3, 0x1
ST:  4, XZCN, .ptext
AS:  S4, 0x29
ST:  5, WIY2CN, .xdata
AS:  S5, 0x2
```

If you are not familiar with the MUFOM commands, you can use the verbose switch. The abbreviated commands such as AS, SA or ST are expanded to *Assignment*, *Section alignment* and *Section type*:

```
pr563 –v –l1 –s calc.out
```

```
ST:  Section type:
     Nbr = 1, type = XAZSN, name =
AS:  Assignment:
     Variable = L1, expression = 0x0
AS:  Assignment:
     Variable = S1, expression = 0x2
ST:  Section type:
     Nbr = 2, type = XZCN, name = .text
AS:  Assignment:
     Variable = S2, expression = 0x3c
.
.
ST:  Section type:
     Nbr = 5, type = WIY2CN, name = .xdata
AS:  Assignment:
     Variable = S5, expression = 0x2
```

The L*n* and S*n* MUFOM variables are defined as the address and the size of section *n*. At level 2 you saw (refer to section 12.7.2.2) that the level 2 view did not mention the L and S variables, because at level 2 the meaning of the L and S variables are known!

Level 0

Switching to level 0 is accomplished by using –l0 (as you expected):

```
pr563 –l0s calc.out
```

```
e6 01 d8 c1 da d3 ce 00
e2 cc 01 81 00
e2 d3 01 02
e6 02 d8 da c3 ce 05 2e 74 65 78 74
e2 d3 02 40
e6 03 d7 c2 d9 02 c3 ce 05 2e 78 62 73 73
e2 d3 03 01
e6 04 d8 da c3 ce 06 2e 70 74 65 78 74
e2 d3 04 29
e6 05 d7 c9 d9 02 c3 ce 06 2e 78 64 61 74 61
e2 d3 05 02
```

The bytes are printed in the MUFOM command structure. It should be easy to find the encoding for the used MUFOM commands. You can use the verbose switch if you want to see file offsets:

```
pr563 –l0vs calc.out
```

```
0000b5   e6 01 d8 c1 da d3 ce 00                          ........
0000bd   e2 cc 01 81 00                                   .....
0000c2   e2 d3 01 02                                      ....
0000c6   e6 02 d8 da c3 ce 05 2e 74 65 78 74              ........text
0000d2   e2 d3 02 40                                      ...@
0000d6   e6 03 d7 c2 d9 02 c3 ce 05 2e 78 62 73 73        ..........xbss
0000e4   e2 d3 03 01                                      ....
0000e8   e6 04 d8 da c3 ce 06 2e 70 74 65 78 74           ........ptext
0000f5   e2 d3 04 29                                      ...)
0000f9   e6 05 d7 c9 d9 02 c3 ce 06 2e 78 64 61 74 61     ..........xdata
000108   e2 d3 05 02                                      ....
```

Viewing Mixed Levels

You can also mix the levels. It is for instance possible to see level 0 and 1 together by specifying option **–l01** (equivalent to **–l10** or **–l0 –l1**):

pr563 –sl01 calc.out

```
ST:  1, XAZSN,
     e6 01 d8 c1 da d3 ce 00
AS:  L1, 0x0
     e2 cc 01 81 00
AS:  S1, 0x2
     e2 d3 01 02
     .
     .
ST:  5, WIY2CN, .xdata
     e6 05 d7 c9 d9 02 c3 ce 06 2e 78 64 61 74 61
AS:  S5, 0x2
     e2 d3 05 02
```

And of course, you can turn on the verbose switch. The switch between level 0 and level 1 is done per MUFOM command. This is because a MUFOM command is the smallest unit at level 1.

If you should display level 1 and 2, the switch is made per object part, because the object parts are the smallest units at level 2. It is not possible to show the results of all section related commands before all these commands are executed:

pr563 –s –l1 –l2 calc.out

```
ST:  1, XAZSN,
AS:  L1, 0x0
AS:  S1, 0x2
.
.
ST:  5, WIY2CN, .xdata
AS:  S5, 0x2
```

```
Section   Size
------------------
Nameless  0x02
.text     0x40
.xbss     0x01
.ptext    0x29
.xdata    0x02
```

12.7.3.3 THE VERBOSE OPTION -v*n*

As you have read in section 12.7.3.2, you can switch to a lower level with the level switch **–l***n*. If you want a verbose printout, you can use the **–v** option.

It is also possible to specify **–v0** to see a verbose output of level 0, option **–v***n* is a shorthand for options **–v –l***n* (or **–vl***n*). The new notation has the advantage that if you want a mixed level output, you are able to choose the verbose option **per level**. You may specify **–l0 –v1**, and you get a non verbose level 0 and a verbose level 1:

```
pr563 –sl0v1 calc.out

ST:  Section type:
     Nbr = 1, type = XAZSN, name =
     e6 01 d8 c1 da d3 ce 00
AS:  Assignment:
     Variable = L1, expression = 0x0
     e2 cc 01 81 00
AS:  Assignment:
     Variable = S1, expression = 0x2
     e2 d3 01 02
.
.
ST:  Section type:
     Nbr = 5, type = WIY2CN, name = .xdata
     e6 05 d7 c9 d9 02 c3 ce 06 2e 78 64 61 74 61
AS:  Assignment:
     Variable = S5, expression = 0x2
     e2 d3 05 02
```

The general verbose switch **–v** (without a number) makes all selected levels verbose. The verbose switch **–v***n* selects level *n* and makes only level *n* verbose.

APPENDIX

A

ASSEMBLER ERROR MESSAGES

TASKING

APPENDIX A

1 INTRODUCTION

The assembler produces error messages on standard error output. If the list option of the assembler is effective, error messages will be included in the list file as well, when the assembler has started list file generation. Error messages have the following layout:

[E|F|W] *error_number*: *filename* line *number* : *error_message*

Example:

as56 E217: /tmp/tst.src line 17 : invalid parallel move

The example reports the error, starting with the severity (E: error, F: fatal error, W: warning) and the error number followed by the source filename and the line number. The last part of the line shows the error message text.

All warnings (W), errors (E), and fatal errors (F) are described below.

2 WARNINGS (W)

The assembler may generate the following warnings:

W 101: ignored ”MODE” directive

The MODE directive is not supported by the DSP56xxx assembler. The assembler always produces relocatable code, except when in an absolute section. See the section *Software Concept* for more information.

W 102: duplicate attribute ”*attribute*” found

An attribute of an EXTERN or an ORG directive is used twice or more. Remove one of the duplicate attributes.

W 103: overlay part of ”ORG” directive not supported

The TASKING DSP56xxx assembler does not support the overlay part of the ORG directive. You can use the locator **lc56** to overlay sections. Remove the overlay definition.

W 104: ”SECTION” attributes ignored

The TASKING DSP56xxx assembler does not support scope–section attributes. All labels defined inside any scope–section are handled as if they were defined using the LOCAL directive, unless they are defined using the EXTERN or GLOBAL directive. Sections are not located according to their scope, but according the ORG directives. See the section *Software Concept* for more information. Remove the SECTION attributes.

W 105: no ”ORG” found yet

Data and program code can only be defined after an ORG directive is used. When a definition is found before an ORG directive is found it will be inserted in a nameless, relocatable P: section. Insert an ORG directive before the offending line.

W 106: conflicting attributes specified ”*attributes*”

You used two conflicting attributes in an EXTERN or an ORG statement directive. For example EXTERN and INTERN. Choose which one you want to use and remove the other.

W 107: memory conflict on object ”*name*”

A label or other object is explicit or implicit defined using incompatible memory types. For example P: and X: memory. Check all usages and definitions of the object *name* to remove this conflict.

W 108: object attributes redefinition ”*attributes*”

A label or other object is explicit or implicit defined using incompatible attributes. For example INTERN and EXTERN. Check all usages and definitions of the object to remove the conflict.

W 109: label ”*label*” not used

The label *label* is defined with the GLOBAL directive and neither defined nor referred, or the label is defined with the LOCAL directive and not referenced. You can remove this label and its definitions (in the case of a LOCAL label).

W 110: extern label ”*label*” defined in module, made global

The label *label* is defined with an EXTERN directive and defined as a label in the source. The label will be handled as a global label. Change the EXTERN definition into GLOBAL or one of the identifiers.

W 111: named section overrules location counter

A named section may not have a location counter. The location counter is ignored. For example:

```
ORG   P1,”.text”:
```

is illegal. Remove the location counter number.

W 112: text found after END, ignored

An END directive designates the end of the source file. All text after the END directive will be ignored. Remove the text.

W 113: named section overrules map attribute

Named sections may not have a mapping (’I’, ’E’, ’R’, ’A’ and ’B’) definition. Use the attribute–names (INTERN and EXTERN) instead. For example:

```
ORG   PI,”.text”:
```

has to be changed into:

```
ORG   P,”.text”,INTERN:
```

W 114: immediate io–short operand not allowed, forced to long

An io–short operator cannot be used in combination with immediate values. The illegal '#<<' operator is changed into a '#>' operator.

W 116: invalid force argument, only NEAR, FAR or NONE allowed

The FORCE and SCSJMP directive only accept NEAR, FAR and NONE as their arguments. Check the supplied argument and change it into one of these.

W 117: reverse–carry buffer and ALIGN directive size expected to be a power of 2

The DSR, BUFFER, BADDR and ALIGN directives only accept sizes which are a power of two. For example: `32` and `128` are valid parameters while `250` is invalid. Check the argument you supplied to the directive and change it in a usable power of two.

W 118: inserted "extern *name*"

The symbol *name* is used inside an expression, but not defined with an EXTERN directive or defined in the current scope. The assembler inserts an EXTERN definition of the offending symbol. Check your label scoping or add an EXTERN definition. You can suppress this message with the 'OPT UR' directive.

W 120: assembler debug information: cannot emit non–tiof expression for *label*

The SYMB record contains an expression with operations that are not supported by the IEEE–695 object format. When the SYMB record is generated by the TASKING C compiler, please fill out the error report and send it to TASKING.

W 121: XDEF interpreted as GLOBAL

The TASKING DSP56xxx assembler changes all XDEF directives into GLOBAL directives. As the semantics of both directives are slightly different the assembler will emit this warning when an XDEF directive is found. Change all XDEF directives to GLOBAL.

W 122: XREF interpreted as EXTERN

The TASKING DSP56xxx assembler changes all XREF directives into EXTERN directives. As the semantics of both directives are slightly different the assembler will emit this warning when an XREF directive is found. Change all XREF directives to EXTERN.

W 123: expression: *type–error*

The expression performs an illegal operation on an address or combines incompatible memory spaces. Check the expression, and change it. Use the @CVS() function to change memory types. You can suppress this message with the 'OPT NOAE' directive.

W 124: memory space modifier unequal to memory space of expression

The memory space modifier used in combination with the expression is not equal to the memory space of the expression result. For example:

```
jmp    p:x_label
```

Check the memory types of the operands, or use the @CVS() function to change the resulting memory space.

W 125: "*symbol*" is not a DEFINE symbol

You tried to UNDEF a symbol that was not previously DEFINEd or was already undefined. Check all DEFINE/UNDEF combinations of the offending symbol.

W 126: redefinition of "*define–symbol*"

The symbol is already DEFINEd in the current scope. The symbol is redefined according to this DEFINE. UNDEF any symbol before redefining it.

W 127: redefinition of macro "*macro*"

The macro is already defined. The macro is redefined according to this macro definition. Purge any macro using PMACRO before redefining it.

W 128: number of macro arguments is less than definition

You supplied less arguments to the macro than when defining it. Check your macro definition with this macro call. The undefined macro arguments are left empty (as in `DEFINE def ''`).

W 129: number of macro arguments is greater than definition

You supplied more arguments to the macro than when defining it. Check your macro definition with this macro call. The superfluous macro arguments are ignored.

W 130: DUPA needs at least one value argument

The DUPA directive needs at least two arguments, the dummy parameter and a value parameter. Add one or more value–parameters.

W 131: DUPF increment value gives empty macro

The step value supplied with the DUPF macro will skip the DUPF macro body. Check the step value.

W 132: IF started in previous file "*file*", line *line*

The ENDIF or ELSE pre–processor directive matches with an IF directive in another file. Check on any missing ENDIF or ELSE directives in that file.

W 133: currently no macro expansion active

The @CNT() and @ARG() functions can only be used inside a macro expansion. Check your macro definitions or expression.

W 134: "*directive*" is not supported, skipped

The directives COBJ, HIMEM, IDENT, LOMEM, LSTCOL, MACLIB, MODE, RDIRECT and SYMOBJ are not supported by the TASKING DSP56xxx assembler. Remove all uses of these directives.

W 135: define symbol of "*define–symbol*" is not an identifier; skipped definition

You supplied an illegal identifier with the **–D** option on the command line. An identifier should start with a letter, followed by any number of letters, digits or underscores.

W 136: expression value outside of fractional domain

The expression resulted in a floating point number less than –1 or greater or equal to 1. The resulting number is saturated.

W 137: label "*label*" defined *attribute* and *attribute*

The label is defined with an EXTERN and a GLOBAL directive. The EXTERN directive is removed, leaving the label global.

W 138: warning: *WARN–directive–arguments*

Output from the WARN directive.

W 139: inserted NOP instruction(s) to remove restriction

The assembler inserted a NOP instruction to accomodate for a pipeline delay. For example:

```
move a,ssh
rts
```

is changed into:

```
move a,ssh
NOP
rts
```

You can toggle the automatic NOP insertion with the OPT directive and on the command line.

W 140: previous instruction sequence may have a pipeline effect

The previous instruction may need to have a NOP instruction between them or, in the case of end of DO loop restrictions, after them.

W 141: *gobal/local* label "*name*" not defined in this module; made extern

The label is declared and used but not defined in the source file. Check the current scope of the label and its usage, change the declaration to EXTERN or add a label definition.

W 142: DO loop target is not a label, cannot check nesting and range

The assembler checks the nesting and range of DO–loops. This cannot be done when the target expression of a DO instruction is not a label. Change the source to use labels instead of address expressions.

W 143: conditional branches to LA are illegal when loop flag is set

This warning is given with Bcc and Jcc branches to LA when the option **–m0** (mask 0F92R and 1F92R) for the DSP563xx assembler (**as563**) is specified.

W 144: missing label for VOID directive; skipped directive

The VOID directive must be preceded by a label.

W 145: more than one cache alignment for this section; changed into "align *number*"

Only one "align cache" may be given per section. When more alignments are needed they can be forced with an align directive, specifying the cache page size as alignment. In code sections the align directive generates NOP instructions to pad the created gap.

W 146: cache alignment may change absolute section origin

The ”align cache” creates a gap at the start of the section to force the next instruction on a cache boundary. This gap is placed before any other part of the section. Therefore, the real section start is at another address than specified with the ORG directive.

W 147: external symbol *”name”* is used with different memory spaces; assuming equate symbol

This is an external which is used with different memory spaces. Assume that it is an equate symbol and tell the user about it. This will not generate a linker warning.

W 148: unrecognized OPT directive *”option”* ignored

Something behind an opt. Whatever is typed there is not recognized.

3 ERRORS (E)

The assembler generates the following error messages when a user error situation occurs. These errors do not terminate assembly immediate. If one or more of these errors occur, assembly stops at the end of the active pass.

E 200: *message*; halting assembly

The assembler stops the further processing of your source file. This is only an informative message. Remove all errors reported earlier and try again.

E 201: unexpected newline or line delimiter

E 202: unexpected character (’*character*’)

The syntax checker found a character that does not confirm to the assembler grammar. Check the line for syntax errors or remove the offending character.

E 203: illegal escape character in string constant

E 205: syntax error: missing *token* before *token*

The syntax checker expected to find a token but found another token. The missing token is inserted before the found token. Check the line for syntax errors.

E 206: syntax error: *token* unexpected

The syntax checker found an unexpected token. The offending token is removed from the input and assembling continues. Check the line for syntax errors.

E 207: syntax error: missing ':'

The syntax checker found a label definition or memory space modifier but missed the appended semi–colon. Check the line for syntax errors, for example misspelled mnemonics.

E 208: syntax error: missing ')'

The syntax checker expected to find a closing parentheses. Check the expression syntax for missing operators and nesting of parentheses.

E 209: invalid radix value, should be '2, '10 or '16

The RADIX directive accepts only 2, 10 or 16.

E 210: syntax error

The syntax checker found an error. Check the line for syntax errors.

E 211: cannot open scope

The given scope–section could not be started. Check the name you supplied with the SECTION directive.

E 212: cannot close scope

The ENDSEC directive could not match with a corresponding SECTION directive. Check your scope nesting.

E 213: label "*label*" defined *attribute* and *attribute*

The label is defined with a LOCAL and a GLOBAL or EXTERN directive. Check your label scoping or change the label declarations.

E 214: illegal addressing mode

The mnemonic used an illegal addressing mode. Check the register usage of address constructs.

E 215: not enough operands

The mnemonic needs more operands. Check the source line and change the instruction.

E 216: too many operands

The mnemonic needs less operands. Check the source line and change the instruction.

E 217: *description*

There was an error found during assembly of the mnemonic. Check the instruction.

E 218: register must be an Rn register

You supplied a register that is not one of R0–R7. Check the instruction.

E 219: register Nn must have same number as Rn

When using (Rn)+Nn, (Rn+Nn) or (Rn)–Nn constructs, the R and the N register must have the same number.

E 220: must be an Rn/Nn register pair

When using (*reg1*)+*reg2*, (*reg1*+*reg2*) or (*reg1*)–*reg2* constructs, *reg1* must be one of R0–R7 and *reg2* must be the corresponding N register.

E 221: multiple scopes with same name

You nested two scope–sections with the same name. Check to see if you didn't forget to close a scope.

E 222: scope not closed

After assembling your entire source the assembler missed some ENDSEC directives. Check all SECTION directives if they have a corresponding ENDSEC directive.

E 224: unknown label "*label*"

The underscore label was used but not defined within the scope of its usage. Check the usage and scope. Try changing the label to a non–underscore label.

E 225: invalid memory type

You supplied an invalid memory modifier with an ORG directive. Check the first character of the offending ORG.

E 226: "E"–memory not supported

The TASKING DSP5600x assembler currently does not support E–memory.

E 227: invalid memory attribute

The assembler found an unknown location counter or memory mapping attribute in an ORG directive.

E 228: more than one location counter

You may only specify one location counter with an ORG directive. Check to see if you specified both ’L’ or ’H’ and a numbered counter.

E 229: location counter must be between 0 and 255

Only 256 different location counters are supported. Change the counter.

E 230: invalid section attribute

The assembler found an unknown ORG attribute. Valid attributes are EXTERN, INTERN, NEAR, FAR, BSS, SCRATCH, OVERLAY, ABSOLUTE, SCRATCH and MAX.

E 231: absolute section, expected expression

When defining an absolute section, you must supply an address–expression after the colon.

E 232: MAX/OVERLAY sections need to be named sections

Sections with the MAX or OVERLAY attribute must have a name.

E 233: code section cannot have *attribute* attribute

Code sections may not have the OVERLAY attribute.

E 234: section attributes do not match earlier declaration

In an previous definition of the same section other attributes were used. Check all section definitions with the same name, try to give only the first ORG of a section all attributes, the later usages only the memory modifier and the name.

E 235: redefinition of absolute section

An absolute section of the same name can only be located once.

E 236: cannot evaluate expression of *descriptor*

Some functions and directives must evaluate their arguments during assembly. Change the expression so that it can be evaluated.

E 237: *descriptor* directive must have positive value

Some directives need to have a positive argument. Check the expression so that is evaluates to a positive number.

E 238: Floating point numbers not allowed with DCB directive

The DCB directive does not accept floating point numbers. Convert the expressions or use the DC directive instead.

E 239: DCB byte constant out of range

The DCB directive stores expressions in bytes. A byte can only hold numbers between 0 and 255.

E 240: DC word constant out of range

The DC directive stores expressions in words. A word can hold 24 bit numbers in X, Y and P space. And 48 bits in L space. Check the range of the expression.

E 241: Cannot emit non tiof functions, replaced with integral value '0'

Floating point expressions and some functions can not be represented in the IEEE–695 object format. When an expression contains unknown symbols it cannot be evaluated and not emitted to the object file. Change these expressions to integral expressions, or make sure they can be evaluated during assembly.

E 242: *directive* directive type must be M(odulo) or R(everse–carry)

The BADDR and BUFFER directives define modulo and reverse carry buffers. Check the arguments you supplied to these directives.

E 243: Nested buffers not permitted

BUFFER directives may not be nested. Check to see if you didn't forget an ENDBUF directive.

E 244: ENDBUF without an BUFFER directive

The assembler found an ENDBUF directive without a corresponding BUFFER directive.

E 245: BUFFER size exceeded

There allocated space between the BUFFER and the ENDBUF command exceeds the defined size of the BUFFER.

E 246: Missing ENDBUF, did you forget to end a buffer?

At the end of the source file a BUFFER directive is still not closed. Check all BUFFER directives.

E 247: illegal condition code

The assembler encountered an illegal condition code within an instruction. Check your input line.

E 248: cannot evaluate origin expression of org ”*name*: *address*”

All origins of absolute sections must be evaluated before creation of the object file. Check the address expression on the usage of undefined or location dependant symbols.

E 249: incorrect argument types for function ”*function*”

The supplied argument(s) evaluated to a different type than expected. Change the argument expressions to the correct type.

E 251: @POS(,,*start*) start argument past end of string

The *start* argument is larger than the length of the string in the first parameter. Change *start* to the correct range.

E 252: second definition of label ”*label*”

The label is defined twice in the same scope. Check the label definitions and rename of remove duplicate definitions.

E 253: recursive definition of symbol ”*symbol*”

The evaluation of the symbol depends on its own value. Change the symbol value exclude this cyclic definition.

E 254: missing closing ’>’ in include directive

The syntax checker missed the closing ’>’ bracket in the include directive. Add a closing ’>’.

E 255: could not open include file *include–file*

The assembler could not open the given include–file. Check the current search path for the presence of the include file and if it may be read.

E 256: integral divide by zero

The expression contains an divide by zero. This is not defined. Change the expression to exclude a division by zero.

E 257: unterminated string

All strings must end on the same line as they are started. Check for a missing ending quot.

E 258: unexpected characters after macro parameters, possible illegal white space

Spaces are not permitted between macro parameters. Check the syntax of the macro call.

E 259: COMMENT directive not permitted within a macro definition and conditional assembly

The TASKING DSP56xxx assembler does not permit the usage of the COMMENT directive within MACRO/DUP definitions or IF/ELSE/ENDIF constructs. Replace the offending COMMENTs with comments starting with a semicolon.

E 260: definition of "*macro*" unterminated, missing "endm"

The macro definition is not terminated with an ENDM directive. Check the macro definition.

E 261: macro argument name may not start with an '_'

MACRO and DUP arguments may not start with an underscore. Replace the offending parameter names with non–underscore names.

E 262: cannot find "*symbol*" in current scope nesting

Could not find a definition of the argument of a '%' or '?' operator within a macro expansion. Check for a definition of the offending symbol.

E 263: cannot evaluate: "*symbol*", value is unknown at this point

The symbol used with a '%' or '?' operator within a macro expansion has not been defined. Insert a definition of the offending identifier.

E 264: cannot evaluate: "*symbol*", value depends on an unknown symbol

Could not evaluate the argument of a '%' or '?' operator within a macro expansion. Check the definition of the offending symbol.

E 265: cannot evaluate argument of *dup* (unknown or location dependant symbols)

The arguments of the DUP directive could not be evaluated. Check the argument expressions on forward references or unknown symbols.

E 266: *dup* argument must be integral

The argument of the DUP directive must be integral. Change the expression so that it evaluates to an integral number.

E 267: *dup* needs a parameter

Check the syntax of the DUP directive.

E 268: ENDM without a corresponding MACRO or DUP definition

The assembler found an ENDM directive without an corresponding MACRO or DUP definition. Check the macro and dup definitions or remove this directive.

E 269: ELSE without a corresponding IF

The assembler found an ELSE directive without an corresponding IF directive. Check the IF/ELSE/ENDIF nesting or remove this directive.

E 270: ENDIF without a corresponding IF

The assembler found an ENDIF directive without an corresponding IF directive. Check the IF/ELSE/ENDIF nesting or remove this directive.

E 271: missing corresponding ENDIF

The assembler found an IF or ELSE directive without an corresponding ENDIF directive. Check the IF/ELSE/ENDIF nesting or remove this directive.

E 272: label not permitted with this directive

Some directives do not accept labels. Move the label to a line before or after this line.

E 273: wrong number of arguments for *function*

The function needs more or less arguments. Check the function definition and add or remove arguments.

E 274: illegal argument for *function*

An argument has the wrong type. Check the function definition and change the arguments accordingly.

E 275: parallel moves are not permitted with this instruction

The instruction does not accept parallel moves. You can use the optimizer to parallelize moves.

E 276: immediate value must be between *value* and *value*

The immediate operand of the instruction does only accept values in the given range. Use the '&' operator to force a value within the needed range or use '#>' to force a long immediate operand.

E 277: address must be between *$address* and *$address*

The address operand is not in the range mentioned. Use the '>' prefix operator to force long addressing or change the address expression.

E 278: operand must be an address

The operand must be an address but has no address attributes. Use an address modifier (e.g. 'X:') or change the address expression.

E 279: address must be short

The operand must be an address in the short range. The expression evaluated to a long address or an address in an unknown range. For the DSP56xxx the addresses between and including P:$0000 and P:$0FFF and for X, Y and L memory $0000 and $003F are in the short range. Use the '<' prefix operator to force the address to a short address or change the address expression.

E 280: address must be short or I/O short

The operand must be an address in the I/O short range. For the DSP5600x the addresses between and including X:$FFC0 and X:$FFFF are legal I/O short addresses. For the DSP563xx the addresses between and including X:$FFFF80 and X:$FFFFFF are legal i/o short addresses. Change the address expression or use the '<<' prefix operator to force the address to the correct I/O short range.

E 281: illegal option "*option*"

The assembler found an unknown or misspelled command line option. The option will be ignored. Use the **–?** option to see a list of all possible options.

E 282: operand must be a Rn or Nn register

An operand of the instruction must be one of the register R0–R7 or N0–N7. Check the instruction and change the operand.

E 283: operand *number* must be *register* register

The referred instruction operand must be one of the mentioned registers. Check the instruction and change its register usage accordingly.

E 284: source and destination must be different

The ADD, SUB, CMP, CMPM, TFR, SUBL, ADDL, SUBR and ADDR instructions must have different source and destination operands. Check the instruction and change one of the operands.

E 285: *file–kind* file will overwrite *file–kind* file

The assembler warns when one of its output files will overwrite the source file you gave on the command line or another output file. Change the name of the source file, use the **–o** option to change the name of the output file or remove the **–err** option to suppress the generation of the error file.

E 286: IC and NOIC options must be given before any symbol definition

The ignore case parameters of the OPT directive may only be given before any symbol is defined. Move the options to the start of the first source file.

E 287: SYMB error: *message*

The assembler found an error in a symbolic debug (SYMB) instruction. When the SYMB instruction is generated by the TASKING C compiler, please fill out the error report form and send it to TASKING. As a work around you could disable the symbolic debug information of this module (remove the **–g** option).

E 288: error in PAGE directive: *message*

The arguments supplied to the PAGE directive do not conform to the restrictions. Check the PAGE directive restrictions in the manual and change the arguments accordingly.

E 289: error in ORG directive

This is an illegal ORG directive or the section name has been used for another, incompatible, section. Check the ORG arguments and the section name.

E 290: fail: *message*

Output of the FAIL directive. This is an user generated error. Check the source code to see why this FAIL directive is executed.

E 291: generated check: *message*

Integrity check for the coupling between the TASKING C compiler and TASKING DSP56xxx assembler. You should not see this error message, unless there are error in user inserted assembly (using the ”#pragma asm” construct).

E 292: no ”ORG” found yet

This error is only generated as part of the integrity checks for the output of the TASKING C compiler. You should not see this error message, unless there are error in user inserted assembly (using the ”#pragma asm” construct).

E 293: expression not in short or I/O short range

An instruction operand must be in the short ($000 through $003F) or I/O short ($FFC0 through $FFFF) address range (for the DSP563xx address range $FFF80 through $FFFFFF). Check the address expression, change it or use the ’<’ or ’<<’ operators to force the operand to the expected type.

E 294: illegal instruction sequence

The previous two instructions may not be executed directly after each other. Insert another instruction or a NOP instruction between them.

E 295: optimizer error: *message*

The optimizer found an error. Try to change the instruction or turn off the the optimizer.

E 296: duplicate destinations are not allowed

An instruction may not have a double write to the same destination register. Change the destination registers or split the instruction in separate moves.

E 297: negative or empty DO loops are not allowed

A DO loop must contain instructions and have an loop address that is after the DO instruction. Insert a NOP in the DO loop body, or change the loop label.

E 298: improper nesting of do loop

When DO loops are nested they must be completely contained inside the outer DO loop.

E 299: jump address must be P in memory

Jumps, jump–subroutines and DO–loops must have a target address in program memory. Check the address expression or use the ’P:’ memory modifier to force the expression into program memory.

E 300: cannot *SCS–action*, no enclosing loop

The structured control statements (SCS) ’.BREAK’ and ’.CONTINUE’ are only allowed inside a loop. Check the loop nesting of your structured control statements.

E 301: missing corresponding ”*SCS–instruction*”

Structured control statements must have proper nesting. Check if all previously started controls have been properly ended.

E 303: unknown condition code ’<*SCS–condition–code*>’

The assembler did not recognize the condition code as a legal DSP56xxx condition code. Legal condition codes are: CC, CS, EC, EQ, ES, GE, GT, HS, LC, LE, LO, LS, LT, MI, NE, NN, NR and PL. Change the condition code to one of these.

E 304: use ”OR” or ”AND” for multiple conditions

The operators ’OR’ and ’AND’ give the relationship between multiple conditions in an structured control expression. Check the syntax of the expression.

E 305: error in structured control expression

The syntax checker found an error in the structured control expression. Check the expression with the syntax description. Common errors are to forget spaces between operands and the ’<’ and ’>’ brackets around condition codes.

E 306: expected conditional operator

The syntax checker found some operands but no logical relation between them add a logical operator, or check if you enclosed the conditional operator between ’<’ and ’>’ brackets.

E 307: expected operand

Conditional structured control operators must have none or two operands. Check the expression.

E 308: use either ”OR” or ”AND”, not both

A structured control expression may only contain either ’OR’ or ’AND’ operations, not both. Change the expression to contain only one of these logical operators.

E 309: cannot combine operands of *statement*, use others or change SCS registers

The assembler must move the operands into the SCS registers to perform assignments and comparisons. As the assembler uses only two scratch registers some combinations of operands can not be transformed into semantically correct assembly. Source operands could be overwritten before before they can be used. Look in the list file for the assembly that is generated and change either the SCS registers with the SCSREG directive or change the operands of the structured control statement.

E 310: illegal combination of operand 1 and 2.

E 311: one of the MOVEP operands must be I/O short

I/O short required for the movep instruction.

E 312: size depends on location, cannot evaluate; probably due to cache alignment

The size of some constructions (notably the align directives) depend on the memory address. The gap for cache alignment can only be calculated when the size of the section is given before the cache alignment directive. When a construction which size depends on the location is placed before the align directive the gap cannot be calculated. Remove the cache alignment directive or change the offending construction.

E 313: cache alignment only valid on P sections

Cache alignment is only sensible on code sections. Remove the align directive.

E 314: ENDM within IF/ENDIF

The assembler found an ENDM directive within an IF/ENDIF pair. Check the MACRO and DUP definitions or remove this directive.

E 315: interrupt section ”*name*” too large for fast interrupts

Name of interrupt section. Fast interrupt has limited size.

E 316: ”Symbols:” part not found in map file *”filename”*

E 317: ”Sections:” part not found in map file *”filename”*

E 318: module *”name”* not found in map file *”filename”*

Name of map file and module. Can occur if is assembler called with map file to generate absolute list file.

E 319: ”*Looplabel*” used as end label for multiple do loops

You used the same loop label for nested loops. Use a different loop label for each nested loop.

E 350: operand *number*; this kind of operand is not permitted here

The referred operand can not be used at that position. Common errors are to forget the '#' operator or an address modifier.

E 351: operand *number* must be in P memory

The referred operand must be an address in program memory. Check the address expression or prefix the operand with 'P:' to force it to program memory.

E 352: operand *number* must be in X or Y memory

The referred operand must be an address in X or Y memory. Check the address expression or prefix the operand with 'X:' or 'Y:' to force it to the wanted memory type.

E 353: illegal operand *number* in parallel Immediate Short Data Move

The referred operand may not be used as a operand of an immediate short data move (move type U). Check the instruction syntax and change the operand or the instruction.

E 354: illegal operand *number* in parallel Long Memory Data Move

The referred operand may not be used as a operand of a long memory data move (move type L). Check the instruction syntax and change the operand.

E 355: illegal *number* operand in *X–or–Y* move field of parallel Register and Y Memory Data Move

The referred operand may not be used as a operand of a register and Y memory data move (move type RY). When the error message refers to the X move field, check the R move, otherwise check the Y move. Change the operand or split the instruction into two separate move instructions.

E 356: illegal operand *number* in parallel Register to Register Data Move

The referred operand may not be used as a operand of a register to register move (move type R). Check the instruction syntax and change the operand or the instruction or change the parallel move to a separate MOVEC instruction.

E 357: illegal operand *number* in parallel Address Register Update Move

The referred operand may not be used as a operand of a register update move (move type U). Only (Rn)–Nn, (Rn)+Nn, (Rn)– and (Rn)+ operands are allowed. Check the instruction syntax and change the operand.

E 358: illegal *number* operand in *X–or–Y* move field of parallel X Memory and Register Data Move

The referred operand may not be used as a operand of a X memory and register data move (move type XR). When the error message refers to the Y move field, check the R move, otherwise check the X move. Change the operand or split the instruction into two separate move instructions.

E 359: illegal *number* operand in *X–or–Y* move field of parallel XY Memory Data Move

The referred operand may not be used as a operand of a X and Y memory data move (move type XY). Check the move field referred to by the error message. Change the operand or split the instruction into two separate move instructions.

E 360: illegal operand *number* in parallel X or Y Memory Data Move

The referred operand may not be used as a operand of a X or Y memory data move (move type X or Y). Change the operand or change the move to a MOVEC instruction.

E 361: no X or Y memory specified on operand *number* in parallel X or Y Memory Data Move

The assembler will try to defer the correct address bus from the expression. When this is not possible you must supply the bus using the 'X:' or 'Y:' memory modifier.

E 362: only (Rn)–Nn, (Rn)+Nn, (Rn)– or (Rn)+ permitted here

Only one of the mentioned operands is permitted. Check the instruction and change the operand accordingly.

E 363: operand must be one of Xn, Yn, An, Bn, A, B, Rn or Nn

Only one of the mentioned registers is permitted. Check the instruction and change the operand accordingly.

E 364: X and Y parallel moves must use different register banks

In a XY memory type move one of the moves must use register R0–R3 and one register R4–R7. Change the registers or split the instruction into two separate moves.

E 365: Bitfield *field–name* out of range

The bit field width is specified by bits 17–12 in S1 register or in immediate control word #CO. The offset from the least significant bit is specified by bits 5–0 in S1 register or in immediate control word #CO. If the offset+width exceeds the value of 56, the result will be undefined.

E 366: operand *number* must be in L memory

Number is 1, 2 or 3. Instruction is VSL instruction, which requires operand in L.

E 368: instruction *name* not supported by DSP56xxx

A couple of instructions are not supported, for instance: norm

4 FATAL ERRORS (F)

The following errors cause the assembler to terminate immediately. Fatal errors are usually due to user errors.

F 401: memory allocation error

A request for free memory is denied by the system. All memory has been used. You may have to break your program down into smaller pieces.

F 402: duplicate input filename "*file*" and "*file*"

The assembler requires one input filename on the command line. Two or more filenames is erroneous.

F 403: error opening *file–kind* file : "*file–name*"

The assembler could not open the given file. When this is a source file, check if the file you specified at the command line exists and if it is readable. When the file is a temporary file, check if the environment symbol TMPDIR has been set correctly.

F 404: protection error : *message*

No protection key or not a IBM compatible PC.

F 405: I/O error

The assembler cannot write its output to a file. Check if you have enough free disk space.

F 407: symbolic debug output error

The symbolic debug information is incorrectly written in the object file. Please fill out the error report form and send it to TASKING.

F 408: illegal operator precedence

The operator priority table is corrupt. Please fill out the error report form and send it to TASKING.

F 409: Assembler internal error

The assembler encountered internal inconsistencies. Please fill out the error report form and send it to TASKING.

F 410: Assembler internal error: duplicate mufom "*symbol*" during rename

The assembler renames all symbols local to a scope to unique symbols. In this case the assembler did not succeed into making an unique name. Please fill out the error report form and send it to TASKING.

F 411: SYMB error: "*message*"

An error occurred during the parsing of the SYMB directive. When this SYMB directive is generated by the TASKING C compiler, please fill out the error report form and send it to TASKING.

F 412: MACRO calls nested too deep (possible endless recursive call)

There is a limit to the number of nested macro expansions. Currently this limit is set to 1000. Check for recursive definitions or try to simplify your source when you encounter this restriction.

F 413: cannot evaluate "*function*"

A function call is encountered although it should have been processed. As a work–around, try to locate the offending function call and remove it from your source. Please fill out the error report form and send it to TASKING.

F 414: cannot recover from previous errors, stopped

Due to earlier errors the assembler internal state got corrupted and stops assembling your program. Remove the errors reported earlier and retry.

F 415: error opening temporary file

The assembler uses temporary files for the debug information and list file generation. It could not open or create one of those temporary files. Check if the environment symbol TMPDIR has been set correctly.

F 416: internal error in optimizer

The optimizer found a deadlock situation. Try to assemble without any optimization options. Please fill out the error report form and send it to TASKING.

F 417: too many errors, stopped

The assembler found too many errors to continue. One error could cause many other errors. Try to solve the first error and assemble again.

F 418: absolute listing file not allowed in combination with –S

Absolute list file generation is not allowed when generating Motorola compatible assembly.

APPENDIX B

LINKER ERROR MESSAGES

TASKING

APPENDIX B

1 INTRODUCTION

Error and warning messages of the linker start with a letter followed by a number and an informational text. The error letter indicates the error type:

W	warning
E	error
F	fatal error
V	verbose message

2 WARNINGS (W)

W 100: Cannot create map file *filename*, turned off –M option

The given file could not be created.

W 101: Illegal filename (*filename*) detected

A filename with an illegal extension was detected.

W 102: Incomplete type specification, type index = T*hexnumber*

An unknown type reference. Arises if a pointer to an unspecified structure is defined.

W 103: Object name (*name*) differs from filename

Internal name of object file not the same as the filename. The file was probably renamed.

W 104: '–o *filename*' option overwrites previous '–o *filename*'

Second **–o** option encountered, previous name is lost.

W 105: No object files found

No files where specified at the invocation.

W 106: No search path for system libraries. Use –L or env "*variable*"

System library files (those given with the **–l** option) must have a search path, either supplied by means of the environment, or by means of the option **–L**.

W 108: Illegal option: *option* (–H or –\? for help)

An illegal option was detected.

W 109: Type not completely specified for symbol <*symbol*> in *file*

Not a complete type specification in either the current file or the mentioned file. This could be an array with unknown depth, or a function with unknown parameters.

W 110: Compatible types, different definitions for symbol <*symbol*> in *file*

Name conflict between compatible types. This could be a member name, tag name for a struct, or a different type name for equal sized basic types (int, long). Note that a basic type conflict is a non portable construct.

W 111: Signed/unsigned conflict for symbol <*symbol*> in *file*

Size of both types is correct, but one of the types contains an unsigned where the other uses a signed type.

W 112: Type conflict for symbol <*symbol*> in *file*

A real type conflict.

W 113: Table of contents of *file* out of date, not searched. (Use **ar ts** <*name*>)

The **ar** library has a symbol table which is not up to date. Generate a new one with '**ar ts**'.

W 114: No table of contents in *file*, not searched. (Use **ar ts** <*name*>)

The **ar** library has no symbol table. Generate one with '**ar ts**'.

W 115: Library *library* contains ucode which is not supported

Ucode is not supported by the linker.

W 116: Not all modules are translated with the same threshold (–G value)

The library file has an unknown format, or is corrupted.

W 117: No type found for <*symbol*>. No type check performed

No type has been generated for the symbol

W 118: Variable <*name*>, has incompatible external addressing modes with file <*filename*>

A variable is not yet allocated but two external references are made by non overlapping addressing modes. This is always an error.

W 119: error from the Embedded Environment: *message*, switched off relaxed addressing mode check

Probably a DELFEE file could not be found. If the embedded environment is readable for the linker, the addressing mode check is relaxed. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.

W 120: Cannot find target description file *name*, relaxed addressing mode check disabled

The linker cannot find the description file (`.dsc`), this means that the linker cannot verify whether addressing modes are compatible.

W 121: Found unresolved external *name*. Setting value to 0.

A symbol was not found. It is filled in the value zero.

3 ERRORS (E)

E 200: Illegal object, assignment of non existing var *var*

The MUFOM variable did not exist. Corrupted object file.

E 201: Bad magic number

The magic number of a supplied library file was not ok.

E 202: Section *name* does not have the same attributes as already linked files

Named section with different attributes encountered. Use **–t** *flag* to see which files are already linked. It is possible that a previously linked file started a .out section with wrong attributes.

E 203: Cannot open *filename*

A given file was not found.

E 204: Illegal reference in address of *name*

Illegal MUFOM variable used in value expression of a variable. Corrupted object file.

E 205: Symbol '*name*' already defined in <*name*>

A symbol was defined twice. The message gives the files involved.

E 206: Illegal object, multi assignment on *var*

The MUFOM variable was assigned more than once probably due to a previous error 'already defined', E205.

E 207: Object for different processor characteristics

Bits per MAU, MAU per address or endian for this object differs with the first linked object.

E 208: Found unresolved external(s):

There were some symbols not found. If **–r** is not set, this is an error.

E 209: Object format in *file* not supported

The object file has an unknown format, or is corrupted.

E 210: Library format in *file* not supported

The library file has an unknown format, or is corrupted.

E 211: Function <*function*> cannot be added to the already built overlay pool <*name*>

The overlay pool has already been built in a previous linker action. Use option **–r** to prevent this.

E 212: Duplicate absolute section name <*name*>

Absolute sections begin on a fixed address. They cannot be linked.

E 213: Section <*name*> does not have the same size as the already linked one

A section with the EQUAL attribute does not have the same size as other, already linked, sections.

E 214: Missing section address for absolute section <*name*>

Each absolute section must have a section address command in the object. Corrupted object file.

E 215: Section <*name*> has a different address from the already linked one

Two absolute sections may be linked (overlaid) on some conditions. They must have the same address.

E 216: Variable <*name*>, *name* <*name*> has incompatible external addressing modes

A variable is allocated outside a referencing addressing space. For instance, the variable was not allocated in the zero page and this variable was referenced with the zero page addressing mode. This is always an error.

E 217: Variable <*name*>, has incompatible external addressing modes with file <*filename*>

A variable is not yet allocated but two external references are made by non overlapping addressing modes. This is always an error.

E 218: Variable <*name*>, also referenced in <*name*> has an incompatible address format

Addresses are often expressed in bytes. In some special cases, the address is expressed in bits. This is necessary for bit variables. An attempt was made to link different address formats between the current file and the mentioned file.

E 219: Not supported/illegal *feature* in object format *format*

An option/feature is not supported or illegal in given object format.

E 220: page size (0x*hexvalue*) overflow for section <*name*> with size 0x*hexvalue*

Section is too big to fit into the page.

E 221: *message*

Error generated by the object. These errors are in fact generated by the assembler. It has been caused by a jump instruction which is out of range.

E 222: Address of <*name*> not defined

No address was assigned to the variable. Corrupted object file.

E 223: Illegal object, empty name assignment on variable *name*

An empty name assignment of a MUFOM variable (type N, X or I).

E 224: Recursive assignment on *name*

A recursion occurred in the assignment of the symbol.

4 FATAL ERRORS (F)

F 400: Cannot create file *filename*

The given file could not be created.

F 401: Illegal object: Unknown command at offset *offset*

An unknown command was detected in the object file. Corrupted object file.

F 402: Illegal object: Corrupted hex number at offset *offset*

Wrong byte count in hex number. Corrupted object file.

F 403: Illegal section index

A section index out of range was detected. Corrupted object file.

F 404: Illegal object: Unknown hex value at offset *offset*

An unknown variable was detected in the object file. Corrupted object file.

F 405: Internal error *number*

Internal fatal error. Passed number will give more information!

F 406: *message*

No key no IBM compatible PC

F 407: Missing section size for section <*name*>

Each section must have a section size command in the object. Corrupted object file.

F 408: Out of memory.

An attempt to allocate more memory failed.

F 409: Illegal object, offset *offset*

Inconsistency found in the object module

F 410: Illegal object

Inconsistency found in the object module at unknown offset.

F 413: Only *name* object can be linked

It is not possible to link object for other processors

F 414: Input file *file* same as output file

Input file and output file cannot be the same.

F 415: Demonstration package limits exceeded

One of the limits in this demo version was exceeded.

F 416: Only one description file allowed

The linker accepts only one description file.

5 VERBOSE (V)

V 000: Abort !

The program was aborted by the user.

V 001: Extracting files

Verbose message extracting file from library.

V 002: File currently in progress:

Verbose message file currently processed.

V 003: Starting pass *number*

Verbose message, start of given pass.

V 004: Rescanning....

Verbose message rescanning library. Rescanning is done if there were new unsatisfied externals during the last scan.

V 005: Removing file *file*

Verbose message cleaning up. Temp files are always removed, map file and .out file are removed if switch –e is on and the exit code is unequal to zero.

V 006: Object file *file* format *format*

Named object file does not have the standard tool chain object format TIOF–695.

V 007: Library *file* format *format*

Named library file does not have the standard tool chain **ar56** format

V 8: Embedded environment *name* read, relaxed addressing mode check enabled

Embedded environment successfully read.

APPENDIX C

LOCATOR ERROR MESSAGES

TASKING

APPENDIX C

1 INTROCUCTION

Error and warning messages of the locator start with a letter followed by a number and an informational text. The error letter indicates the error type:

W	warning
E	error
F	fatal error
V	verbose message

2 WARNINGS (W)

W 100: Maximum buffer size for *name* is *size* (Adjusted)

For the given format, a maximum buffer size is defined.

W 101: Cannot create map file *filename*, turned off –M option

The given file could not be created.

W 102: Only one –g switch allowed, ignored –g before *name*

Only one .out file can be debugged.

W 104: Found a negative length for section *name*, made it positive

Only stack sections can have a negative length.

W 107: Inserted '*name*' keyword at line *line*

A missing keyword in the description file was inserted.

W 108: Object name (*name*) differs from filename

Internal name of object file not the same as the filename. Maybe renamed?

W 110: Redefinition of system start point

Usually only one load module will access the system table (__lc_pm).

W 111: Two –o options, output name will be *name*

Second **–o** option, the message gives the effective name.

W 112: Copy table not referenced, initial data is not copied

If you use a copy statement in the layout part, the initial data is located in rom. Your start–up code should copy this data to their ram location.

W 113: No .out files found to locate

No files where specified at the invocation.

W 114: Cannot find start label *label*

No start point found.

W 116: Redefinition of *name* at line *line*

Identifier was defined twice.

W 119: File *filename* not found in the argument list

All files to be located must be given as an argument.

W 120: unrecognized *name* option <*name*> at line *line* (inserted '*name*')

Wrong option assignment. Check the manual for possibilities.

W 121: Ignored illegal sub–option '*name*' for *name*

An illegal format sub option was detected. See the format description for this format in the manual.

W 122: Illegal option: *option* (–H or –\? for help)

An illegal option was detected.

W 123: Inserted *character* at line *line*

The given character was missing in the description file.

W 124: Attribute *attribute* at line *line* unknown

An unknown attribute was specified in the description file.

W 125: Copy table not referenced, blank sections are not cleared

Sections with attribute blank are detected, but the copy table is not referenced. The locator generates info for the startup module in the copy table for clearing blank sections at startup. See __lc_cp in the manual.

W 127: Layout *name* not found

The used layout in the named file must be defined in the layout part.

W 130: Physical block *name* assigned for the second time to a layout

It is not possible to assign a block more than once to a layout block.

W 136: Removed *character* at line *line*

The character is not needed here.

W 137: Cluster *name* declared twice (layout part)

The named cluster is declared twice. Duplicate cluster names are allowed in the layout part under conditions, because the clusters are referred only. In the layout part the cluster is declared, which may be done only once.

W 138: Absolute section *name* at non–existing memory address 0x*hexnumber*

Absolute section with an address outside physical memory. Either the address is not correct, or the memory description for your target is not consistent.

W 139: *message*

Warning message from the embedded environment. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.

W 140: File *filename* not found as a parameter

All processes defined in the locator description file (software part) must be specified on the invocation line.

W 141: Unknown space <*name*> in –S option

An unknown space name was specified with a –S option.

W 142: No room for section *name* in read–only memory, trying writable memory ...

A section with atribute read–only could not be placed in read–only memory, the section will be placed in writable memory.

W 143: Section *name*s has different page size than previous group members

Section has a different page size then other sections in the same group.

W 144: Filename *name* is too long, truncated to *name*

Filename is too long and is truncated.

W 145: Conflicting output options c (chip level) and s (start record), s ignored

Output sub–options ’s’ and ’c’ are conflicting sub–options. The s option is ignored.

W 146: Address width in output format (*number* bytes) is too small for address *address*(hex). Only first occurrence reported.

The width of the address format is too small to contain the complete address.

W 147: Conflict between absolute section address 0x*hexaddress* and alignment *number* specified in the description file (alignment ignored)

In the description file an alignment is specified for a section, which conflicts with the absolute address of the section. The alignment is ignored.

W 148: Conflict between section alignment *number*, and address 0x*hexaddress* specified in the description file (alignment will be applied to address)

In the description file an absolute address is added to a section, this address conflicts with the alignment of the section. The alignment will be applied to the address before locating.

3 ERRORS (E)

E 200: Absolute address 0x*hexnumber* occupied

An absolute address was requested, but the address was already occupied by another section.

E 201: No physical memory available for section *name*

An absolute address was requested, but there is no physical memory at this address.

E 202: Section *name* with mau size *size* cannot be located in an addressing mode with mau size *size*

A bit section cannot be located in a byte oriented addressing mode.

E 203: Illegal object, assignment of non existing var *var*

The MUFOM variable did not exist. For some variables this is an error.

E 204: Cannot duplicate section '*name*' due to hardware limitations

The process must be located more than once, but the section is mapped to a virtual space without memory management possibilities.

E 205: Cannot find section for *name*

Found a variable without a section, should not be possible.

E 206: Size limit for the section group containing section *name* exceeded by 0x*hexnumber* bytes

Small sections do not fit in a page any more.

E 207: Cannot open *filename*

A given file was not found.

E 208: Cannot find a cluster for section *name*

No writable memory available, or unknown addressing mode. Often this error occurs due to an error in the description file.

E 210: Unrecognized keyword <*name*> at line *line*

An unknown keyword was used in the description file.

E 211: Cannot find 0x*hexnumber* bytes for section *name* (fixed mapping)

One of virtual or physical memory was occupied, or there was no physical memory at all!

E 213: The physical memory of *name* cannot be addressed in space *name*

A mapping failed. There was no virtual address space left.

E 214: Cannot map section *name*, virtual memory address occupied

An absolute mapping failed. The memory on the virtual target address was already occupied.

E 215: Available space within *name* exceeded by *number* bytes for section *name*

The available addressing space for an addressing mode has been exceeded.

E 217: No room for section *name* in cluster *name*

The size of the cluster as defined in the .dsc file is too small.

E 218: Missing *identifier* at line *line*

This identifier must be specified.

E 219: Missing ')' at line *line*

Matching bracket missing.

E 220: Symbol '*symbol*' already defined in <*name*>

A symbol was defined twice.

E 221: Illegal object, multi assignment on *var*

The MUFOM variable was assigned more than once, probably due to an error of the object producer.

E 223: No software description found

Each input file must be described in the software description in the .dsc file.

E 224: Missing <length> keyword in block '*name*' at line *line*

No length definition found in hardware description.

E 225: Missing <*keyword*> keyword in space '*name*' at line *line*

For the given mapping, the keyword must be specified.

E 227: Missing <start> keyword in block '*name*' at line *line*

No start definition found in hardware description.

E 230: Cannot locate section *name*, requested address occupied

An absolute address was requested, but the address was already occupied by another process or section.

E 232: Found file *filename* not defined in the description file

All files to be located need a definition record in the description file.

E 233: Environment variable too long in line *line*

Found environment variable in the dsc file contains too many characters.

E 235: Unknown section size for section *name*

No section size found in this .out file. In fact a corrupted .out file.

E 236: Unrecoverable specification at line *line*

An unrecoverable error was made in the description file.

E 238: Found unresolved external(s):

At locate time all externals should be satisfied.

E 239: Absolute address *addr.addr* not found

In the given space the absolute address was not found.

E 240: Virtual memory space *name* not found

In the description files software part for the given file, a non existing memory space was mentioned.

E 241: Object for different processor characteristics

Bits per MAU, MAU per address or endian for this object differs with the first linked object.

E 242: *message*

Error generated by the object. These errors are in fact generated by the assembler. It has been caused by a jump instruction which is out of range.

E 244: Missing *name* part

The given part was not found in the description file, possibly due to a previous error.

E 245: Illegal *name*value at line *line*

A non valid value was found in the description file

E 246: Identifier cannot be a number at line *line*

A non valid identifier was found in the description file

E 247: Incomplete type specification, type index = T*hexnumber*

An unknown type was referenced by the given file. Corrupted object file.

E 250: Address conflict between block *block1* and *block2* (memory part)

Overlapping addresses in the memory part of the description file.

E 251: Cannot find 0x*hexnumber* bytes for section *section* in block *block*

No room in the physical block in which the section must be located.

E 255: Section '*name*' defined more than once at line *line*

Sections cannot be declared more than once in one layout/loadmod part.

E 258: Cannot allocate reserved space for process *number*

The memory for a reserved piece of space was occupied.

E 261: User assert: *message*

User–programmed assertion failed. These assertions can be programmed in the layout part of the description file.

E 262: Label '*name*' defined more than once in the software part

Labels defined in the description file must be unique.

E 264: *message*

Error from the embedded environment. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.

E 265: Unknown section address for absolute section *name*

No section address found in this .out file. In fact a corrupted .out file.

E 266: *funcionality* not (yet) supported

The requested functionallity is not (yet) supported in this release.

E 267: Absolute section at address 0x*hexaddress* does not fit in page.

The absolute section crosses the specified page boundary.

4 FATAL ERRORS (F)

F 400: Cannot create file *filename*

The given file could not be created.

F 401: Cannot open *filename*

A given file was not found.

F 402: Illegal object: Unknown command at offset *offset*

An unknown command was detected in the object file. Corrupted object file.

F 403: Illegal filename (*name*) detected

A filename with an illegal extension was detected on the command line.

F 404: Illegal object: Corrupted hex number at offset *offset*

Wrong byte count in hex number. Corrupted object file.

F 405: Illegal section index

A section index out of range was detected. This could be a corrupted object file, but also a previous error like E231 (Missing section) is responsible for this message.

F 406: Illegal object: Unknown hex value at offset *offset*

An unknown variable was detected in the object file. Corrupted object file.

F 407: No description file found

The locator must have a description file with the description of the hardware and the software of your system.

F 408: *message*

No protection key or not an IBM compatible PC.

F 410: Only one description file allowed

The locator accepts only one description file.

F 411: Out of memory.

An attempt to allocate more memory failed.

F 412: Illegal object, offset *offset*

Inconsistency found in the object module.

F 413: Illegal object

Inconsistency found in the object module at unknown offset.

F 415: Only *name* .out files can be located

It is not possible to locate object for other processors.

F 416: Unrecoverable error at line *line*, *name*

An unrecoverable error was made in the description file in the given part.

F 417: Overlaying not yet done

Overlaying is not yet done for this .out file, link it first without **–r** flag!

F 418: No layout found, or layout not consistent

If there are syntax errors in the layout, it may occur that the layout is not usable for the locator. Syntax errors in the description file must be resolved!

F 419: *message*

Fatal from the embedded environment. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.

F 420: Demonstration package limits exceeded

One of the limits in this demo version was exceeded.

F 421: Error writing file *name*

An error occurred when writing to the file.

F 422: Input file *name* same as output file

Input file and output file cannot be the same.

5 VERBOSE (V)

V 000: File currently in progress:

Verbose message. On the next lines single filenames are printed as they are processed.

V 001: Output format: *name*

Verbose message for the generated output format.

V 002: Starting pass *number*

Verbose message, start of given pass.

V 003: Abort !

The program was aborted by the user.

V 004: Warning level *number*

Verbose message, report the used warning level.

V 005: Removing file *file*

Verbose message cleaning up. Temporary files are always removed, map file and .out file are removed if switch **–e** is on and the exit code is unequal zero.

V 006: Found file <*filename*> via path *pathname*

The description (include) file was not found in the standard directory. The locator searches also in the install directory `etc`, in which the file was found.

V 007: *message*

Verbose message from the embedded environment. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.

APPENDIX D

ARCHIVER ERROR MESSAGES

TASKING

D APPENDIX

1 INTRODUCTION

This appendix contains all warnings (W), errors (E) and fatal errors (F) of the archiver **ar56**.

2 WARNINGS (W)

W 100: Illegal warning level: *level*

Warning level is a single digit.

W 101: Member *name* not found

Library member not found, warning only.

W 102: Can't modify modification time for *name*

The archiver cannot access the file *name* to change the modification time.

W 103: creating archive *name*

The **q** option was used while archive file did not exist (**r** option would be more appropriate).

W 104: Option –a or –b only allowed with key option 'r' or 'm'. Ignored!

Option **a** or **b**, which specifies a position in the archive can only be applied with replace or move actions.

W 105: Only one position specification allowed, ignored '–a or –b *file_offset*'

It is not possible to specify more than one position in the archive. The options **–a** and **–b** are both used to specify a position.

W 106: Option –o only allowed with key option 'x'. Ignored!

Library date can only be preserved with extraction of a library member.

W 107: Option –u only allowed with key option 'r'. Ignored!

Objects newer than the archive are only replaced with key option r.

W 108: Option –z only allowed with key option 'r'. Ignored!

Only objects which are moved to the archive can be checked.

W 109: Option –v has no meaning with key option 'p' or 't'. Ignored!

For options p and t the verbose switch is meaningless.

W 110: Option –s may be used only with option –t.

W 111: Illegal symbol level: *level*

Symbol level is a single digit.

W 112: Name *name* is too long, truncated to *name*

The name exceeded the limit, and is truncated.

3 ERRORS (E)

E 200: filename too long

The filename was too long to fit into the internal buffer.

E 201: Member *name* not found

Library member not found.

E 204: Can't obtain file–status information *filename*

Cannot access *filename* to obtain file status information.

E 207: illegal option: *option*

An illegal option was detected.

E 209: Can't rename file: *name* to: *name*

Renaming the library file to a tempfile failed.

4 FATAL ERRORS (F)

F 300: user abort

The library manager is aborted by the user.

F 301: too much errors

The maximum number of errors is exceeded.

F 302: protection error: *error*

error message received from ky_init.

F 303: can't create "*filename*"

Cannot create the file with the mentioned name.

F 304: can't open "*filename*"

Cannot open the file with the mentioned name.

F 305: can't reopen '*filename*'

The file *filename* could not be reopened.

F 306: read error while reading "*filename*"

A read error occurred while reading named file.

F 307: write error

A write error occurred while writing to the output file.

F 308: out of memory

An attempt to allocate memory failed.

F 309: illegal character

A character which is not allowed was found.

F 310: *filename* not in archive format

the archive file given is not in the proper format.

F 311: specification of more than one key {rxdmpt} is not permitted

More than one key was given.

F 312: no one of the keys {rxdmpt} was specified

No key was given.

F 313: error in the invocation. Use option –? or –H to get help.

Show usage. For more help, use option **–?**.

F 314: *name* does not exist

Library will only be created in case the r key–option is specified.

F 315: IEEE violation for object module *name* at address *address*

IEEE violation detected (**z** option enabled).

F 316: corrupted object module *name*

The object module name does not conform to the IEEE object specification.

F 317: *name*: illegal byte count in hex number, offset = *offset*

Illegal byte count in hex number (IEEE violation).

F 318: evaluation date expired !!

APPENDIX

EMBEDDED ENVIRONMENT ERROR MESSAGES

E

TASKING

E APPENDIX

1 INTRODUCTION

Error and warning messages from the embedded environment are part of the linker and/or locator error messages. The error numbers mentioned below are not part of the message.

E	error
W	warning

2 ERRORS (E)

E 1: Conflicting attributes *attributes* at line *number*

Conflicting attributes.

E 2: Unknown attribute '*character*' at line *number*

Unknown attribute.

E 3: Unknown keyword '*name*' at line *number*

Unknown keyword.

E 4: Illegal character '*character*' at line *number*

Illegal character.

E 5: Page size only allowed in a space definition at line *number*

Page size only allowed in space definition.

E 6: Page size must be a power of 2 at line *number*

Page size must be a power of 2.

E 7: Mau size must be a power of 2 at line *name*

Mau size must be a power of 2.

E 8: Cannot synchronize any more line *number*

Cannot synchronize any more.

E 9: Illegal value '*value*' at line *number*

Illegal value.

E 10: Illegal hex value '*value*' at line *number*

Illegal hex value.

E 11: Illegal octal value '*value*' at line *number*

Illegal octal value.

E 12: Missing value at line *number*

Missing value.

E 13: Illegal identifier at line *number*

Illegal identifier.

E 14: Wrong attribute '*attribute*' at line *number*

Attribute not allowed.

E 15: Unknown identifier '*name*' at line *number*

Unknown identifier.

E 16: Inserted '*character*' at line *number*

Inserted character.

E 17: Cannot find bus/space '*name*' in definition for space '*name*'

Error in the destination of mapping from space.

E 18: Cannot find space/amode '*name*' in definition for amode '*name*'

Map error.

E 19: Cannot find chip '*name*' in definition for bus '*name*'

Map error.

E 20: Cannot find space/amode '*name*' in layout definition for segment '*name*'

Map error.

E 21: Cannot find bus '*name*' in definition for mapping '*name*'

Map error.

3 WARNINGS (W)

W 100: Cannot find mapping '*name*' in segment definition for space '*name*'

Warning in segment mapping.

W 101: Section '*name*' should be defined in amode '*name*', not amode '*name*'

The section was specified in the wrong addressing mode

APPENDIX F

MIGRATION FROM MOTOROLA CLAS

TASKING

APPENDIX F

1 INTRODUCTION

This appendix explains how you can migrate your assembly program from the Motorola CLAS assembler to the TASKING DSP56xxx assemblers (**as56**, and **as563**). It also describe the implementation differences between the TASKING DSP56xxx assembler, linker and locator and the Motorola CLAS assembler and linker/locator.

The TASKING assemblers are source compatible with the Motorola CLAS assembler. However, there are some exceptions to this rule. Most notable are the differences in the overlay and scoping (SECTION directive) support. These and other exceptions are discussed below. The TASKING assemblers also introduce some new features that are not supported by the CLAS assembler. Most notable are new keywords and the acceptance of forward references in almost all expressions. The TASKING assemblers are capable of performing optimizations like move parallelization and instruction reordering to adjust for pipelining restrictions.

2 ABSOLUTE AND RELATIVE MODE

The TASKING assembler always produces relocatable code. The behavior of the sections (SECTION directive) is the same as with the Motorola CLAS assembler in absolute mode.

3 OBJECT FORMAT

The TASKING assemblers generate object files that conform to the IEEE–695 object format. This format has no support for floating point numbers and name scopes. Therefore, relocatable expressions may not contain floating point expressions, and these expressions (when defined with equate directives) cannot be exported to other modules using the GLOBAL directive. The assembler will emit an error message on emitting floating point expressions. When it is necessary to share floating point expressions between different modules it is possible to define these expressions using an equate directive in a file, and include that file in all necessary modules. As the object format does not support name scopes, as introduced by the SECTION directive, non–global symbols defined within scopes (i.e. enclosed in a SECTION/ENDSEC pair) must be renamed to symbols on the global module level. How this is done, and when you can use these symbols from other modules, is documented in section 3.4 *Scopes*.

4 ASSEMBLER DIRECTIVES

4.1 UNSUPPORTED DIRECTIVES

The following directives are not supported by the TASKING assemblers (**as56** and **as563**). The assemblers issue a warning when an unsupported directive is found in the input file and the directive is ignored.

Unsupported CLAS–specific directives (because IEEE–695 object format does not support them):

COBJ

IDENT

SYMOBJ

Other unsupported assembler directives:

HIMEM and LOMEM directives

The memory bounds are defined in the locator description file. The scheme offered by the locator is much more flexible.

RDIRECT directive

It is not possible to remove mnemonic or directive names from the symbol table of the assembler. However, it is possible to define preprocessor symbols (with DEFINE or MACRO) that have the same name as mnemonic or directive names.

MACLIB directive

Macro libraries are not supported.

MODE directive

The assembler always operates in relocatable mode.

XDEF directive

This directive is obsolete, the assemblers treat this directive as a GLOBAL directive.

XREF directive

This directive is obsolete, the assemblers treat this directive as an EXTERN directive.

4.2 CHANGED DIRECTIVES

The following Motorola CLAS directives have a slightly different behavior in the TASKING assemblers:

COMMENT directive

This directive is not permitted in IF/ELSE/ENDIF constructs and MACRO/DUP definitions.

INCLUDE directive

This directive does not assume a default extension of ".asm". You should supply it yourself. The default extension behavior gives rise to unclearness which file is included.

EQU/SET/GSET directives

These directives accept forward references.

FORCE/SCSJMP directives

These directives accept the modes SHORT and LONG for CLAS compatibility only. The preferred modes are now NEAR and FAR, as they are used by the directives ORG and EXTERN. These are the same attributes as used in the C compiler.

LOCAL and GLOBAL directive

These directives are also permitted in the module body.

OPT directive

Not all options of the OPT directive are supported due to the different architecture of the TASKING assemblers. Options that are not recognized are ignored.

The following options are not supported:

FC	FF	FM	PP	RC	
CEX	CL	CRE	DXL	HDR	
IL	LOC	MC	NL	S	U
DEX	NS	SCL	SCO	SO	XR
CK	CM	CONST	CONTCHCK		
DLD	GL	GS	INTR	LB	LDB
MI	PSM	RSV	SI		

ORG directive

The syntax of the ORG directive is extended for type checking. The overlay specification part of the ORG directive is not supported. When it is supplied the assembler issues a warning. Specification of the overlaying must be done at locate time.

SECTION directive

Section are only used for scoping of symbols. They are not the basis for code ordering and relocation. We strongly encourage the use of the module concept as explained in chapter 3 *Software Concept*. The GLOBAL, STATIC and LOCAL attributes are not accepted. Use the GLOBAL and LOCAL directives to define the scope attributes for individual symbols.

4.3 NEW DIRECTIVES

The following directives are new with respect to the Motorola CLAS assembler:

ALIGN	–	specify alignment
EXTERN	–	declare extern symbols
SYMB	–	pass high level language debug information to the object file
CALLS	–	pass call information to object file. Used to build a call tree at link time for overlaying overlay sections.
VOID	–	Control DO loop optimization.

5 STRUCTURED CONTROL STATEMENTS

The assemblers do an extensive job for producing semantically legal data move code when the SCS statements are used. As the CLAS assembler does not do this, the TASKING assemblers will produce different code. It will be necessary to check the code generated by the SCS statements on register usage.

6 SECTIONS AND OVERLAYING

The CLAS assembler overlays sections by defining attributes to the ORG directive (using runtime and load location counters). The TASKING assemblers overlay sections using a different, more flexible, strategy. In the paragraph 3.3 *Sections* the overlay mechanism is explained in detail.

7 ASSEMBLER FUNCTIONS

The following assembler mode functions are not supported:

@CCC(), @CHK(), @CTR(), @EXP(), @LCV(), @REL(), @NSR(), @INT()

8 EXPRESSIONS

Expressions are typed with respect to the semantics of the different operators. Possible types are: address (with attributes to denote the memory space and address range), floating–point, integer and string. Therefore, the TASKING assemblers may type complex expressions differently than the CLAS assembler. Use the conversion functions to resolve possible typing conflicts.

9 FORWARD REFERENCES

The CLAS assembler does not accept forward references in expressions. The TASKING assemblers do accept forward references in almost every expression. Exceptions are expressions that are used to reserve memory space, like DS and BSM, and expressions that are used as parameters to preprocessor directives, like IF and DUP. The expressions supplied with these kind of directives must be evaluated before the assembler can proceed. Therefore, they may not contain forward references.

10 OPTIMIZATIONS

The TASKING assemblers can perform optimizations to the supplied assembly program. When this is requested (using the **–O** command line parameter, or the **OPT OP** directive) the supplied source is changed. The new instruction combinations and ordering can be examined in the list file, and by using the **pr56** utility. The list file is annotated which source line is moved to which place and which source line is combined with another instruction. This can be very complex.

APPENDIX G

DESCRIPTIVE LANGUAGE FOR EMBEDDED ENVIRONMENTS

TASKING

APPENDIX G

1 INTRODUCTION

In an embedded environment an accurate description of available memory and control over the behavior of the locator is crucial for a successful application. For example, it may be necessary to port applications to processors with different memory configurations, or it may be necessary to tune the location of sections to take full advantage of fast memory chips.

For this purpose the DELFEE language, which stands for DEscriptive Language For Embedded Environments, was designed.

2 GETTING STARTED

2.1 INTRODUCTION

This section gives a general introduction about the DELFEE description language. The goal is to give you an overview and some basic knowledge what the DELFEE description language is about, and how a basic description file looks. A more detailed description and examples are given in the following sections.

2.2 BASIC STRUCTURE

The DELFEE language describes where code or data sections should be placed on the actual memory chips. This language has to define the interface between a virtual world (the software) and a physical world (the hardware configuration).

On the one side, in the virtual world, there are the code and data sections which are described by the assembly language. Sections can have names, attributes like writable or read–only and can have an address in the addressing space or an addressing mode describing the range of the address space in which they may be located.

On the other side, the physical world, the actual processor is present which reads instructions from memory chips and interprets these instructions. With the DELFEE language you can instruct the locator to place the code and data sections at the correct addresses, taking into account things like the type of memory chip (rom/ram, fast/slow), availability of memory, etc. The DELFEE language gives the possibility to tune the same application for different hardware configurations.

In the DELFEE language the interface between virtual and physical world is described in three parts:

1. **software part** (`*.dsc`)

   The software part belongs to the virtual world and describes the order in which data and code sections should be located. The software part may vary for different applications and can even be empty.

2. **cpu part** (`*.cpu`)

   The cpu part is the interface between the virtual world and the real world. It contains the application independent part of the virtual world (the address translation of addressing modes to the addressing space), and the configuration independent part of the physical world (on–chip memory, address busses). The cpu part is independent of application and configuration.

3. **memory part** (`*.mem`)

   The memory belongs to the physical world. It contains the description of the external memory. The memory part may vary for different configurations and can even be empty (if there is no external memory).

The software part and the memory part can be empty, but the cpu part must always be defined.

The DELFEE language is used in a special file, which is called the description file. In the DELFEE description language the different parts are defined with the following syntax:

```
software {
     layout {
          // ordering of sections
     }
}

cpu {
     // mapping of addressing modes to address space
     // defining address space
     // mapping of address space to actual busses
     // defining on–chip memory
}

memory {
     // description of external memory
}
```

You can use C++ style comments. Everything after '//' until the end of line is ignored.

For convenience the cpu part and the memory part can be placed in different files, which makes it possible to have different layout parts for different applications and different memory parts for different configurations. The files can be included using the syntax:

```
cpu filename     // include cpu part defined in file filename
mem filename     // include memory part defined in file filename
```

3 CPU PART

3.1 INTRODUCTION

The cpu part contains the application and configuration independent part of the description file. This part defines the translations of the addresses from the assembler language (virtual addresses) all the way down to the chips (physical addresses). To describe the translations, DELFEE recognizes four main levels:

1. addressing mode(s) definitions. Addressing modes are subsets of an address space. They define address ranges within an address space.
2. address space(s) definitions. The address space is the total range of addresses available.
3. bus(ses) definitions.
4. (on–chip) memory chips definitions.

The address translation is defined from addressing mode via space and bus to the chip. The addressing modes and the busses can be nested, the space and the chip cannot.

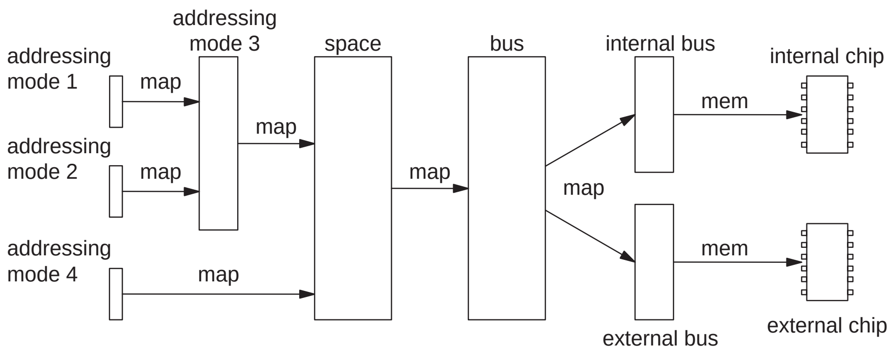


Figure G–1: Address translation

The addressing modes and addressing spaces belong to the virtual part, the busses and chips belong to the physical part. The following sections describe the address space and the addressing modes which are subsets of the address space. Then a description of the physical side (hardware configuration) follows, describing the busses and chips that are available.

The following example illustrates what a cpu part could look like. It is a fictitious example, mainly used to illustrate the definitions. You should be able to recognize the addressing mode definitions, address space definition, bus definitions and on–chip memory definition. Each definition is explained in the following sub–sections.

```
cpu {
   //
   // addressing mode definitions
   //
   amode near_code {
      attribute Y1;
      mau 8;
      map src=0 size=1k dst=0 amode = far_code;
   }
   amode far_code {
      attribute Y2;
      mau 8;
      map src=0 size=32k dst=0 space = address_space;
   }
   amode near_data {
      attribute Y3;
      mau 8;
      map src=0 size=1k dst=0 amode = far_data;
   }
   amode far_data {
      attribute Y4;
      mau 8;
      map src=0 size=32k dst=32k space = address_space;
   }

   //
   // space definitions
   //
   space address_space {
      mau 8;
      map src=0   size=32k dst=0   bus = address_bus label = rom;
      map src=32k size=32k dst=32k bus = address_bus label = ram;
   }

   //
   // bus definitions
   //
   bus address_bus {
      mau 8;
      mem addr=0   chips=rom_chip;
      map src=0x100 size=0x7f00  dst=0x100 bus = external_rom_bus;
      mem addr=32k chips=ram_chip;
      map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
   }
   //
   // internal memory definitions
   //
   chips rom_chip  attr=r mau=8 size=0x100;  // internal rom
   chips ram_chip  attr=w mau=8 size=0x100;  // internal ram
}
```

3.2 ADDRESS TRANSLATION: MAP AND MEM

In DELFEE there are two ways to describe a memory translation between two levels (the source level and the destination level):

1. **map** keyword. This is for address translations between amodes, spaces, busses (not chips).
2. **mem** keyword. This describes the address translation between bus and chip. **mem** is a simplified case of **map**.

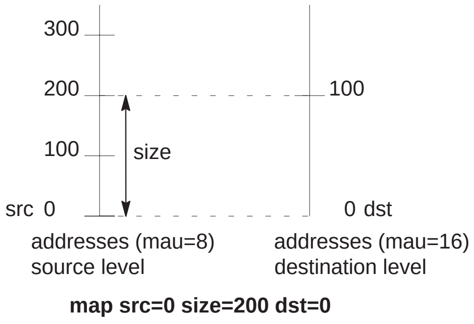


Figure G–2: Map address translation

The generalized syntax for the map definition is (see figure G–2):

map **src=***number* **size=***number* **dst=***number*
*destination_type***=***destination_name* *optional_specifiers***;**

where,

src	start address of the source level. In case of an address translation between amodes and spaces, the source level is the amode and the destination level is the space.
size	length of the source level.
dst	start address at the destination level.

destination_type the destination type depends on the context the mapping is used in and can have three different types:

1. **amode** allowed in context: amode.
2. **space** allowed in context: amode.
3. **bus** allowed in context: space, bus.

optional_specifiers The optional identifiers are also dependent of the context they are used in:

1. **label** Only allowed in space context and needed as a reference for the block definition in the software part (see section 4.5).

   label = *name* **;**

2. **align** This indicates that every section will be aligned at the specified value.

   align = *number* **;**

3. **page** This indicates that every section should be within a given page size.

   page = *number* **;**

Both the source level and the destination level have an address range that is expressed in a number of Minimum Addressable Units (MAU, the minimal amount of storage, in bits, that is accessed using an address). The mapping only describes the range and the destination of the address mapping, the actual transformation also depends on the memory unit that an address can access. If a source level with a minimum addressable unit of 8 bits (`mau=8`) maps to a destination level with a minimum addressable unit of 16 bits (`mau=16`), the size of the destination level, expressed in address range, is half the original size. So, according to figure G–2, the size of the destination level is 100.

If a map is present from *level1* down to *level2*, the map definition works as follows:

$$\mathit{end\_address\ of\ level2} = \mathit{dst} + (\ \mathit{size} * \mathit{mau\ of\ level1}\ /\ \mathit{mau\ of\ level2}\ )$$

The **mem** description is actually a simplified case of the **map** description. The length of the address translation is taken from the chip size, the destination address is always zero. It is used to map a bus to a chip.

The syntax is:

mem addr=*number* **chips=***name***;**

where,

addr	start address location of a chip.
chips	the name of the chip that is located at address *number*.

3.3 ADDRESS SPACES

The link between the virtual and the physical world is the description of the address space and the way it maps onto the internal address busses.

The address space is defined by the complete range of addresses that the instruction set can access. Some instruction sets support multiple address spaces (for example a data space and a code space).

An address space is described by the syntax:

```
space name {
    mau number;
    map src=number  size=number dst=number bus=bus_name label=name;
        //    :
        // more maps
}
```

where,

space	defines the name by which the space can be referenced in the description file.
mau	the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

map this specifies the mapping of a range of addresses in the address space to a bus defined by *bus_name*. The range of addresses is defined by **src** and **size**, the offset on the bus is defined by **dst**. (The bus you map the address space on, may have a different MAU, which will lead to another length of the range of the bus). An address space can only map onto a bus.

Usually an address in the address space corresponds to the same address on the bus. In that case **src** and **dst** have the same value.

In the previous example there is one space definition:

```
space address_space {
   mau 8;
   map src=0   size=32k dst=0   bus = address_bus label = rom;
   map src=32k size=32k dst=32k bus = address_bus label = ram;
}
```

In this example the space is named `address_space`. Note that the **amode** definitions use this name as destination for their mappings. The minimum addressable unit (MAU) is set to 8 bits. The labels `rom` and `ram` are used by **block** definitions in the software part which are discussed in section 4.5.

3.4 ADDRESSING MODES

Addressing modes define address ranges in the addressing space. Addressing modes usually have a special characteristic, like bitaddressable part of memory, parts especially for code sections, zero pages, etc. The addressing modes are defined by the instruction set. The syntax of defining an addressing mode in the DELFEE language is:

An address space is described by the syntax:

amode *name* **{**
  mau *number***;**
  attr Y*number***;**
  map src=*number* **size=***number* **dst=***number* **amode|space=***name***;**
}

where,

amode The name by which the addressing mode can be referenced. In the object file the addressing mode of a section is encoded with an **Y***number*. This means that the *name* given to the addressing mode has only meaning within the description file, not to the sections!

mau the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

attr Y the addressing mode number. Code or data sections (generated by the assembler) all have a number specifying the addressing mode they belong to. In the DELFEE description file this number is used to identify the addressing mode. This number must never be changed, because the interpretation of the sections will get mixed up.

map defines the mapping of the addressing mode to another addressing mode (**amode**) or an address space (**space**).

Below is an example of two addressing mode definitions:

```
amode near_data {
     attribute Y3;
     mau 8;
     map src=0 size=1k dst=0 amode = far_data;
}
amode far_data {
     attribute Y4;
     mau 8;
     map src=0 size=32k dst=32k space = address_space;
}
```

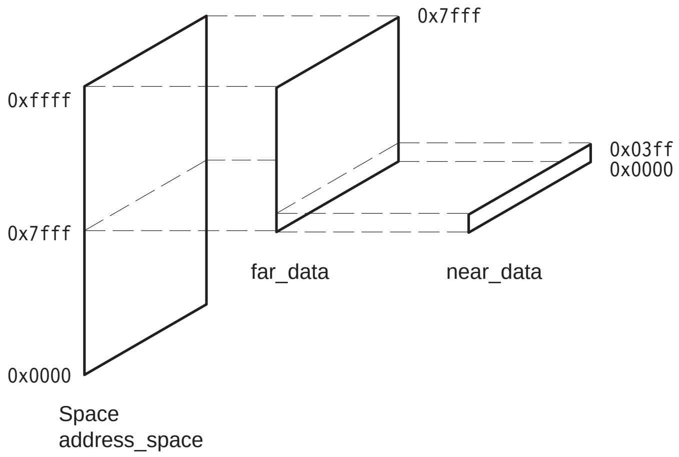


Figure G–3: Addressing mode mapping

In this example the addressing modes are named `near_data` and `far_data`. They are identified by the addressing mode numbers `Y3` and `Y4` respectively. The minimum addressable unit (MAU) is set to 8 bits. Addressing mode `near_data` maps on addressing mode `far_data`, and `far_data`, in its turn, maps on address space `address_space`. address_space is the space as discussed in the previous section.

3.5 BUSSES

The **bus** keyword describes the bus configuration of a cpu. In essence it describes the address translation from the address space to the chip. The syntax is:

bus *name* {
    mau *number*;
    map src=*number* **size=***number* **dst=***number* **bus=***name*;
    mem addr=*number* **chips=***name***;**
}

where,

bus	the name by which the bus can be referenced.
mau	the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

map	mapping to another bus.
mem	mapping to a memory chip.

Below is an example of a bus definition:

```
bus address_bus {
    mau 8;
    mem addr=0   chips=rom_chip;
    map src=0x100 size=0x7f00  dst=0x100 bus = external_rom_bus;
    mem addr=32k chips=ram_chip;
    map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
}
```

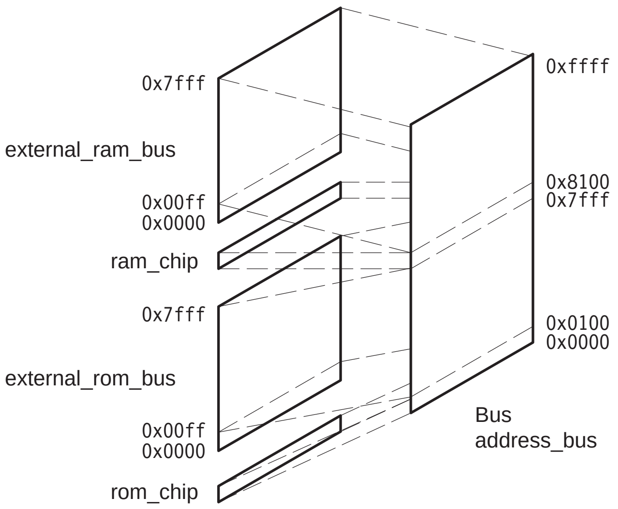


Figure G–4: Bus mapping

In this example the address bus is named `address_bus`. The minimum addressable unit (MAU) is set to 8 bits. The internal memory chip `rom_chip` is located at address 0 of the bus, and the chip `ram_chip` is located at address `32k`.

Two address mappings to other busses are present: one to `external_rom_bus` and one to `external_ram_bus`.

The first mapping translates addresses `0x100–0x7ff` of `address_bus` (`src=0x100 size=0x7f00`) onto addresses of `external_rom_bus` starting at address `0x100` (`dst=0x100`).

The second mapping translates addresses `0x8100–0xffff` of `address_bus` (`src=0x8100 size=0x7f00`) onto addresses of `external_ram_bus` starting at address `0x100` (`dst=0x100`).

The second mapping maps to RAM, not ROM. That is why both destination addresses are the same.

3.6 CHIPS

The **chips** keyword describes the memory chip. The syntax is:

chips *name* **attr=***letter_code* **mau=***number* **size=***number***;**

where,

chips	the name by which the chip can be referenced.
attr	defines the attributes of the chip with a letter code
letter_code	one of the following attributes:

r	read–only memory.
w	writable memory.
s	special memory (it must not be located).

mau	the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.
size	the size of the chip (address range from 0–*size*).

Below is an example of two chip definitions:

```
chips rom_chip  attr=r mau=8 size=0x100;  // internal rom
chips ram_chip  attr=w mau=8 size=0x100;  // internal ram
```

In this example the chips are named `rom_chip` and `ram_chip`. The minimum addressable unit (MAU) is set to 8 bits. The size of both chips is `0x100` MAUs (= 256 bytes). Chip `rom_chip` is read–only and chip `ram_chip` writable, as you would expect with ROM and RAM.

3.7 EXTERNAL MEMORY

With the syntax described in the previous sections it would be possible to define mappings from an address space to external memory chips (DELFEE does not actually know, or care, if memory is on–chip). However, this is not advisory. For maintenance and flexibility reasons it is better to keep the internal (static) memory part apart from the external (variable) memory part. The chapter *Memory Part* describes how to deal with external memory.

In the cpu part you only have to define a mapping to an external bus, which can later be defined in the memory part. The following example contains references to two external busses: `external_ram_bus` and `external_rom_bus`.

```
bus address_bus {
   mau 8;
   mem addr=0   chips=rom_chip;
   map src=0x100 size=0x7f00  dst=0x100 bus = external_rom_bus;
   mem addr=32k chips=ram_chip;
   map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
}
```

4 SOFTWARE PART

4.1 INTRODUCTION

The software part has two main parts:

1. **load_mod**
2. **layout description**

```
software {
     load_mod start = start_label;

     layout {
          // ordering of sections
     }
}
```

4.2 LOAD MODULE

The keyword **load_mod** defines the program start label. The program start label is the start of the code and the reset vector should point to this label. The locator generates a warning if this label is not referenced.

```
load_mod start = start_label;
```

4.3 LAYOUT DESCRIPTION

First of all, the layout definition can be omitted. If you omit the layout definition, the locator will generate a layout definition based on the DELFEE description of the amodes (addressing modes) in the cpu part (See section 3). However this does not allow you to control the order in which sections (like stack and heap) are located. If you define the layout part, the locator uses this description.

The layout part is probably the most difficult part of the DELFEE language. It is designed to give the locate algorithm the information it needs to locate the sections correctly. Through some examples you will be shown how to control the locating process using the DELFEE language.

To give you an idea of where all this will lead to, an example of a layout part is given:

```
layout {
      space address_space {
            block rom {
                  cluster first_code_clstr {
                        attribute i;
                        amode near_code;
                        amode far_code;
                  }
                  cluster code_clstr {
                        attribute r;
                        amode near_code {
                              section selection=x;
                              section selection=r;
                        }
                        amode far_code {
                              table;
                              section selection=x;
                              section selection=r;
                              copy;    // locate rom copies here
                        }
                  }
            }
            block ram {
                  cluster data_clstr {
                        attribute w;
                        amode near_data {
                              section selection=w;
                        }
                        amode far_data {
                              section selection=w;
                              heap;
                              stack;
                        }
                  }
            }
      }
}
```

The layout definition is defined with the syntax:

layout {
 // space definitions
}

The first thing to notice is the different levels inside the layout definition:

space This level can only occur inside a layout level. There are as many space levels as there are space definitions in the cpu part.

block	This level can only occur inside a space level. There are as many block levels as there are mappings defined in the space definition in the cpu part.
cluster	This level can only occur inside a block level. There can be multiple clusters inside a block. Their main purpose is to group (code/data) sections. The locator locates each cluster in the specified order.
amode	This level can only occur inside a cluster level. An **amode** corresponds to an **amode** definition in the cpu part. Within an **amode** you can specify the order in which data/code sections are located.

The four levels can roughly be divided in two groups. The **space** and **block** definition correspond to address ranges and the **cluster** and **amode** definition correspond to (groups of) sections.

The following paragraphs first introduce the **space** and **block** definition. Then separate paragraphs show how to select certain groups of sections and how this is used in the **cluster** and **amode** definition.

4.4 SPACE DEFINITION

Section 3.3 already defined the address translation of a space in the cpu part. In the example in that section, the following space was defined:

```
space address_space {
   mau 8;
   map src=0   size=32k dst=0   bus = address_bus label = rom;
   map src=32k size=32k dst=32k bus = address_bus label = ram;
}
```

For every space defined in the cpu part you have to provide a description in the layout definition.

The space level should be inside the layout definition and can only contain one or more block levels.

The name of the space must correspond to a space definition in the cpu part.

The syntax is:

```
space  name {
    // block definitions
}
```

Below is an example of a space definition from the software part:

```
space address_space {
     block rom {
          ....
     }
     block ram {
          ...
     }
}
```

In this example space `address_space` defines two blocks: block `rom` and block `ram`.

4.5 BLOCK DEFINITION

With the **block** description you can set boundaries to the sections based on chip sizes.

A block references a physical area of memory. Selected sections are only allowed within the range of the block description. In effect a block limits the range in which a section can be located.

The physical address range of a block is actually defined in the cpu part by a labeled mapping:

```
space address_space {
   mau 8;
   map src=0   size=32k dst=0   bus = address_bus label = rom; //<--
      // --> block name: rom
   map src=32k size=32k dst=32k bus = address_bus label = ram; //<--
      // --> block name: ram
}
```

The name of the **block** description must correspond to a label in the **map** definition of a **space** definition in the cpu part. The **block** definition must be inside the **space** definition and can only contain one or more cluster levels.

The syntax is:

block *name* {
  // cluster definitions
}

Below is an example of a bus definition from the software part:

```
block rom {
     cluster first_code_clstr {
          ...
     }
     cluster code_clstr {
          ...
     }
}
```

In this example block `rom` defines two clusters: cluster `first_code_clstr` and cluster `code_clstr`.

4.6 SELECTING SECTIONS

The previous paragraphs explained how the address ranges are defined by block definitions, now it is time to select the sections that should be placed in these blocks. In DELFEE there are two levels in which you can define the order of locating:

1. **cluster**
2. **amode**

To define the locating order you need to have some kind of handle to specify a section or a group of sections. DELFEE recognizes the following characteristics of a section:

name of the section	This is unique to a specific section.
attribute(s) of a section	The attributes of a section are specified by the assembler or compiler. Possible attributes are defined in table G–1. By selecting an attribute you select a group of sections. The attributes can be grouped to an attribute string, for example: **by1w**.

addressing mode All sections have an addressing mode (as defined in the cpu part).

attr	**Meaning**	**Description**
W	Writable	Must be located in ram
R	Read only	Can be located in rom
X	Execute only	Can be located in rom
Y*num*	Addressing mode	Must be located in addressing mode *num*
A	Absolute	Already located by the assembler
B	Blank	Section must be initialized to ’0’ (cleared)
F	Not filled	Section is not filled or cleared (scratch)
I	Initialize	Section must be initialized in rom
N	Now	Section is located before normal sections (without N or P)
P	Postponed	Section is located after normal sections (without N or P)

Table G–1: Section Attributes

To specify a (group) of sections, DELFEE has the following syntax:

1. select a group on section attribute:

   section selection = *attr***;**

2. select a section by name:

   section *name***;**

3. select a special section:

   heap; //locate heap here
   stack; //locate stack here
   table; //locate copy table here
   copy; //locate all initial data here
   copy *name***;**//locate initial data of the named section here

4. create a section:

   reserved label=*name* **length=***number***;**

Instead of selecting a section by an attribute, DELFEE also allows excluding a section by its attribute.

Excluding an attribute is done by placing a '**–**' (minus sign) in front of *attr*.

So, the example:

```
section selection=attr1–attr2
```

selects a group of sections with attribute *attr1* and without attribute *attr2*.

4.7 CLUSTER DEFINITION

Clusters are used to place specified sections in a group. The locator will handle the clusters in the order that they are specified. This gives you the possibility to create a group of selected sections and give it a higher locate priority.

There are several possibilities to specify that a section is part of a cluster. The exact rules and their priorities are given in the paragraph *Section Placing Algorithm*. The three main possibilities are:

1. attribute
2. section selection=
3. amode definition

Examine the following example:

```
layout {
      space address_space {
            block rom {
                  cluster first_code_clstr {
                        attribute i;
                        amode near_code;
                        amode far_code;
                  }
                  cluster code_clstr {
                        attribute r;
                        amode near_code {
                              section selection=x;
                              section selection=r;
                        }
                        amode far_code {
                              table;
                              section selection=x;
                              section selection=r;
```

```
                            copy;    // locate rom copies here
                        }
                    }
                }
            }
}
```

In this example an extra cluster `first_code_cluster` was created. Using the placing algorithm (paragraph 4.10) you can see that sections with attribute ’i’ will be placed in cluster `first_code_clstr` and therefore will get a higher priority than sections in cluster `code_clstr`.

The syntax is:

> **cluster** *name* **{**
>   // section selections
> **}**

Within a cluster the sections with the least freedom are located first. Freedom is defined by the possible addresses a section can be located at.

4.8 AMODE DEFINITION

Within a cluster you can specify an addressing mode or amode. Although in the cpu part (paragraph 3.4) an address range was assigned to every amode, in the layout part the addressing mode is used to identify groups of sections.

The syntax is:

>   :
> **amode** *name* **{**
>   **section selection =** *attr***;**
>   :
> **}**
>   :

The order of locating is now determined by the order of specification.

For example, suppose you want to locate all writable sections first, then the heap, followed by the stack. In the DELFEE language this is specified by:

```
      :
   section selection = w;   // 'w' means writable sections
   heap;
   stack;
      :
```

4.9 MANIPULATING SECTIONS IN AMODES

The previous paragraphs explained how to set the order of the sections within an **amode** definition. DELFEE recognizes an extra set of keywords to further tune the locating of code and data sections.

An **amode** definition can contain the following keywords:

Keyword	Description
section	Selects a section, or group of sections
selection	Specifies attributes for grouping sections
attribute	Assigns attributes (are past to the cluster
copy	Selects a rom copy of a section by name, or all rom copies in general
fixed	Forces a section to be located around a fixed address
gap	Creates a gap in the address range where sections will not be located
reserved	Reserves a memory area, which can be referenced using locator labels
heap	Defines the place and attributes of the heap
stack	Defines the place and attributes of the stack
table	Defines the place and attributes of the copy table
assert	A user defined assertion
length	Specifies the length of stack, heap, physical block or reserved space

Table G–2: amode keywords

All keywords are described in section 7, *Delfee Keyword Reference*.

4.10 SECTION PLACING ALGORITHM

There are different ways to reference a section. Sections can be referenced as a group based on a certain attribute, or they can be referenced very specific by name. To find out where sections are placed in the layout part, DELFEE uses the following algorithm:

1. First, try to find a selection by section name.
2. If not found, search for a ’section selection=’ within a matching amode block.
3. If not found, search for a ’section selection=’ not within an amode block.
4. If not found, search for a cluster with a correct ’amode= ..,..,.. ;’ and correct attributes.
5. If not found, search for a cluster with correct attributes.
6. If not found, relax attribute checking, and start over again.

Relax attributes using the following rules:

1. If stack, heap or reserved, switch indication off and try again.
2. If attribute ’f’ (not filled), switch ’f’ off and try again.
3. If attribute ’b’ (clear), switch ’b’ off and try again.
4. If attribute ’i’ (initialize), switch ’i’ off and try again.
5. If attribute ’x’ (executable code), switch ’x’ off and ’r’ (read–only) on and try again. (Try to place executable sections in read–only memory).
6. If attribute ’r’ (read–only), switch ’r’ off ’w’ (writable) on and try again. (Try to place read–only sections in writable memory).

5 MEMORY PART

5.1 INTRODUCTION

The memory part defines the variable part of the memory configuration. It can be placed in a different file, which allows to easily switch between different memory configurations. The syntax used for the mappings is the same as used in the cpu part.

As you have seen in the example of the cpu part in section 3, there were two references to external busses:

```
bus address_bus {
   mau 8;
   mem addr=0   chips=rom_chip;
   map src=0x100 size=0x7f00  dst=0x100 bus = external_rom_bus;
   mem addr=32k chips=ram_chip;
   map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
}
```

In the memory part you have to define the description for the busses `external_rom_bus` and `external_ram_bus`. Using the description in sections 3.5 and 3.6 for specifying busses and chips, the memory part could look like:

```
memory {
   bus external_rom_bus {
      mau 8;
      mem addr=0 chips=xrom;
   }

   chips xrom attr=r mau =8 size=0x8000;

   bus external_ram_bus {
      mau 8;
      mem addr=0 chips=xram;
   }

   chips xram attr=w mau=8 size=0x8000;
}
```

6 DELFEE PREPROCESSING

6.1 INTRODUCTION

You can preprocess a DELFEE description file using exactly the same syntax as used by the C preprocessor. This means that all preprocessor directives start with a '#'–sign.

The preprocessor scans the input (description) file looking for macro calls. A macro–call is a request to the preprocessor to replace the call pattern of a built–in or user–defined macro with its definition.

There are two types of macro definitions: 'plain' macros and 'function–like' macros. A plain macro is expanded to a fixed string of characters. A function–like macro looks like a function call. The macro is expanded to its definition, in which the macro parameters are replaced by their co corresponding macro arguments.

6.2 USER DEFINED MACROS

You can create macros with the **#define** preprocessor directive.

Syntax:

#define *macro–name*[**(***formal–parameter–list***)**] *macro–body*

When you create a parameterless macro, there are two parts to a **#define** call: the *macro–name* and the *macro–body*. The *macro–name* defines the name used when the macro is called; the *macro–body* defines the return value of the call.

The *macro–body* is usually the return value of the macro call. However, the macro–body may contain calls to other macros. If so, the return value is actually the fully expanded macro–body, including the return values of the call to other macros.

Example:

```
#define ASIZE  10
```

Every occurrence of `ASIZE` is expanded to '10'.

If the only function of the macro processor was to perform simple string replacement, then it would not be very useful for the most programming tasks. Each time you want to change even the simplest part of the macro's return value you would have to redefine the macro. Parameters in macro calls allow more general–purpose macros. Parameters leave holes in a macro–body that are filled in when you call the macro. This permits you to design a single macro that produces code for typical operations. The term 'parameters' refers to both the formal parameters that are specified when the macro is defined (the holes), and the actual parameters or argument that are specified when the macro is called (the fill–ins). To define macros with parameters you have to add a *formal–parameter–list*. The *formal–parameter–list* is a list of macro identifiers separated by ','. These identifiers comprise the formal parameters used in the macro. The macro identifier for each parameter in the list must be unique.

Example:

After

```
#define ADD(a, b) a + b
```

the call `ADD(4, 5)` is expanded to `4 + 5`.

You can undefine a preprocessor macro with the **#undef** preprocessor directive:

#undef *macro–name*

6.3 FILE INCLUSION

With the **#include** preprocessor directive:

#include <*include–file***>**

you can include text from *include–file* within the input text of the description file. At the occurrence of an **#include** control line, the preprocessor reads the text from *include–file* until end–of–file is reached. **#include** files may be nested. *include–file* is any file that contains description file information. include–file is searched for in the directory `etc` directory relative to the installation path of your product.

The ANSI standard defines the following terms for the include directive:

#include ”*include–file*”
#include **<***include–file***>**
#include *token–sequence*

The preprocessor uses the following search rules for include files between ” ”:

1. search in the directory of the description file
2. search in the directory `etc` relative to the installation path of your product

Note that if you nest include files, the preprocessor applies the first rule for each level.

Example:

```
product.dsc:
     #include ”../inc/product.cpu”

product.cpu:
     #include ”prod2.cpu”
```

According to rule 1 the preprocessor searches `prod2.cpu` in the same directory as `product.cpu` since `prod2.cpu` is included by product.cpu and not by `product.dsc`.

The preprocessor searches for include files between **< >** in the same way as for include files between **” ”**. The difference is that rule 1 does not apply (the directory of the source description file is not searched).

The third form of include directives:

#include *token–sequence*

means that the included file name may be a token sequence that has been defined before. After expansion by the preprocessor this should produce a valid include directive as described by the first two forms:

Example:

```
#ifdef STD
#define cpu_incl <product.cpu>
#else
#define cpu_incl ”my_cpu.cpu”
#endif

#include cpu_incl
```

6.4 CONDITIONAL STATEMENTS

Some preprocessor directives expect logical *expressions* in their arguments. Logical *expressions* follow the same rules as numeric *expressions*. The difference is in how preprocessor interprets the value that the *expression* represents. Once the *expression* has been evaluated to a value, the preprocessor uses the '= 0' comparison to determine whether the expression is TRUE or FALSE (if the value is equal 0 the expression is FALSE else TRUE).

The **#if** and **#elif** preprocessor directives evaluate a logical *expression*, and based on that *expression*, expand or withhold their *statements*. The **#ifdef** and **#ifndef** preprocessor directives evaluates the existence of a user–defined macro, and based on the result, expand or withhold their *statements*.

Syntax:

if–line
    statements
[**#elif** *expression*
    statements]...
[**#else**
    statements]
#endif

where *if–line* is one of:

#if *expression*
#ifdef *macro–name*
#ifndef *macro–name*

The *expression* in the **#if** directive and subsequent **#elif** directives are evaluated in order until a TRUE value is encountered. If the value is TRUE, then the preprocessor expands the succeeding *statements*; if the value is FALSE and the optional **#else** directive is included in the call, then the *statements* succeeding **#else** are expanded. If the *expression* results to FALSE and the **#else** is not included, the **#if** call returns the null string.

The **#ifdef** tests if the *macro–name* is a previously defined macro. The **#ifndef** evaluates its *statements* if the *macro–name* is not currently defined.

Each **#if**, **#ifdef** and **#ifndef** directive must have a corresponding **#endif.**

Example:

```
#define _STCK 100
#if _STCK == 100
     stack length=100;
#else
     stack length=200;
#endif
```

This example always expands to: `stack length=100;`. In this case the **#if** control line could also be written as:

```
#ifdef _STCK
```

7 DELFEE KEYWORD REFERENCE

This section contains an alphabetical description of all keywords that can be used in a description file. Some keywords can be abbreviated to a minimum of four characters. Everything after '//' until the end of line is considered a comment.

.addr

Syntax:

.addr (Software part)

Description:

The predefined label **.addr** contains the current address.

Example:

```
block ram {
 cluster data_clstr {
  attribute w;
  amode near_data {
   section selection=w;
   assert ( .addr < 256, ”page overflow”);
     // if the condition is false,
     // the locator generates an error with
     // the text as message
  }
  ...
 }
}
```

address

Syntax:

address = *address* (all parts)
addr = *address* (abbreviated form)

Description:

Specify an absolute address in memory.

Example:

Cpu or memory part:

```
bus address_bus {
     mau 8;
     mem addr=0   chips=rom_chip;
     ...
     mem addr=32k chips=ram_chip;
     ...
}
```

Software part:

```
block rom {
     ...
     cluster code_clstr {
          attribute r;
          amode near_code {
               section selection=x;
               section selection=r;
               section .string address = 0x0100;
          }
          ...
     }
}
```

The locate order in the **amode** definition in the example above is fixed. Sections with attribute selection 'x' and/or 'r' are forced to be located before section `.string`. If this fixed order is not desired, the absolute address specification can be done in a separate **amode** definition.

Example:

```
amode near_code {
     section .string address = 0x0100;
}

amode near_code {
     section selection=x;
     section selection=r;
}
```

align

Syntax:

align = *power_of_2* (Software part)

Description:

Specify the alignment of a section. The alignment should be a power of 2 (even addresses).

Example:

In the following example section `DATA` will be aligned on 2 MAUs:

```
...
amode near_data {
     section DATA align = 2;
}
...
```

amode

Syntax:

(Cpu or memory part)
amode *identifier*[**,** *identifier*]... **{** *amod_description* **}** (def)
amode = *identifier* (ref)

amode *identifier*[**,** *identifier*]... **;** (Software part)
amode *identifier*[**,** *identifier*]... **{** *section_blocks* **}**

Description:

The keyword **amode** can appear in all parts. In the cpu or memory part you can use **amode** to map an addressing mode or register bank on a particular address space (definition). When you specify **amode=**, you map a specific addressing mode on a previously defined addressing mode (reference). The only keywords allowed in an *amod_description* (cpu part) are **attribute**, **map** and **mau**. The keyword **attribute Y***num* uniquely identifies the addressing mode.

In the software part you can use **amode** as part of a cluster definition to change the locating order of sections. See also 4.10, *Section Placing Algorithm*.

Example:

From cpu or memory part:

```
cpu {
  amode near_data {
      attribute Y3;
      mau 8;
      map src=0 size=1k dst=0 amode = far_data;
           // reference
  }
  amode far_data { // definition
      attribute Y4;
      mau 8;
      map src=0 size=32k dst=32k space = address_space;
  }
```

From software part:

```
block ram {
  cluster data_clstr {
     attribute w;
     amode near_data {
       // Sections with addressing mode
       // near_data are located here
       section selection=w;
     }
     amode far_data {
       // Sections with addressing mode
       // far_data and the stack and heap
       // are located here
       section selection=w;
       heap;
       stack;
     }
  }
}
```

assert

Syntax:

assert (*condition* **,** *text* **) ;** (Software part)
asse (*condition* **,** *text* **) ;** (abbreviated form)

Description:

Test condition of virtual address in memory. Generate an error if the assertion fails and give a message with '*text*'. *condition* is specified as one of:

expr1 **>** *expr2*
expr1 **<** *expr2*
expr1 **==** *expr2*
expr1 **!=** *expr2*

expr1 and *expr2* can be any expression or label. The predefined label **.addr** contains the current address.

Example:

```
block ram {
 cluster data_clstr {
  attribute w;
  amode near_data {
   section selection=w;
   assert ( .addr < 256, ”page overflow”);
     // if the condition is false,
     // the locator generates an error with
     // the text as message
  }
  ...
 }
}
```

attribute

Syntax:

attribute *attribute_string* **;**	(Software part)
attr *attribute_string* **;**	(abbreviated form)
attribute = *attribute_string*	(Software part)
attr = *attribute_string*	(abbreviated form)

Description:

With **attribute** you can assign attributes to sections, clusters or memory blocks. See also the keyword **selection**.

For sections these attributes are pure supplementary to the standard section attributes. The standard section attributes such as zero page (Y1), blank (B) and executable (X) are set by the compiler (or by the assembler in the case of an assembler program).

With an action attribute after a section (**attr=**), you can set section attributes or you can disable section attributes with the **–** (minus) sign.

The attributes have the following meaning:

num	(Section only) Align the section at 2^{num} MAUs.
Y*num*	(amode and sections only) Identify addressing mode. Indicate that sections with this attribute should be allocated in this cluster.
r	(Memory and clusters) Indicate this is a read–only cluster or read–only memory.
w	(Memory and clusters) Indicate this is a writable cluster or writable memory.
s	(Memory only) Indicate this is special memory, it must not be located.
x	(Clusters/sections only) Indicate that the cluster/section is executable.
b	(Clusters/sections only) Indicate that clusters/sections should be cleared before locating.

i	(Sections only) Indicate that clusters/sections should be copied from ROM to RAM.
f	(Clusters/sections only) Indicate that clusters/sections should not be filled and not cleared. This is called a scratch cluster/section.

Default attributes if the attribute keyword is omitted:

sections:	The attributes as generated from the assembler/compiler.
clusters:	The attributes as indicated by the underlaying memory, thus **r** for rom and **w** for ram.
memory:	If no attributes defined, the default is writable (**w**).

Example:

From software part:

```
layout {
   space address_space {
     block rom {
       cluster first_code_clstr {
          attribute i; // set cluster attribute
          amode near_code;
          amode far_code;
       }
     }
```

```
      block ram
        cluster ram {
           amode near_data {
             // Default attribute of cluster
             // data is ’w’, because the
             // memory is RAM.

            section selection=w;
            section selection=b attr=-b;
             // Sections with attribute b are
             // are located here, and
             // attribute ’b’ is switched off
           }
           .
        }
        .
      }
    }
 }
```

From cpu part:

```
   amode near_data {
       attribute Y3;  //identify code with Y3
       mau 8;
       map src=0 size=1k dst=0 amode = far_data;
   }
   ...

    chips rom_chip  attr=r mau=8 size=0x100;
    chips ram_chip  attr=w mau=8 size=0x100;
       ...
       // memory attributes
```

block

Syntax:

block *identifier* { *block_description* } (Software part)

Description:

With **block** you define the contents of a physical area of memory. You can make a **block** description for each chip you use. Each block has a symbolic name as previously defined by the keyword **chips**. It is allowed to combine two or more memory chips in one block as long as their total address range is linear, without gaps. The identifier indicates that a memory block starts at the specified chip, no matter how many chips are combined.

Example:

```
layout {
  space address_space {
     block ram
      // Memory block starting at chip ram_chip
       cluster ram {
          ...
       }
     }
  }
}
```

bus

Syntax:

(Cpu or memory part)

bus *identifier*[**,** *identifier*]... **{** *bus_description* **}** (def)

bus = *identifier*[**|** *identifier*]... (ref)

Description:

With **bus** you define the physical memory addresses for the chips that are located on the cpu (definition). When you specify **bus=**, you map a specific address range on a previously defined address bus (reference). You can provide parallel busses by separating each bus with a vertical bar '**|**'. The only keywords allowed in an **bus** description are **mem**, **map** and **mau**.

Example:

```
cpu {
   space address_space {
      // Specify space 'address_space' for the address_bus
      // address bus.
      mau 8;
      map src=0   size=32k dst=0   bus = address_bus label = rom;
      map src=32k size=32k dst=32k bus = address_bus label = ram;
                                   // ref
   }

   bus address_bus {  // definition
      mau 8;
      mem addr=0   chips=rom_chip;
      map src=0x100 size=0x7f00  dst=0x100 bus = external_rom_bus;
      mem addr=32k chips=ram_chip;
      map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
   }
   ...
}
```

chips

Syntax:

(Cpu or memory part)

chips *identifier*[**,** *identifier*]... *chips_description* (def)
chips = *identifier*[**|** *identifier*]... [**,** *identifier*[**|** *identifier*]...]...
(ref)

Description:

With **chips** you describe the chips on the cpu or on your target board (definition). For each chip its **size** and minimum addressable unit (**mau**) is specified. With the keyword **attr** you can define if the memory is read–only. The only three attributes allowed are **r** for read–only, **w** for writable, or **s** for special. If omitted, w is default.

You can use **chips=** after the keyword **mem** to specify where a chip is located (reference). You can create chip pairs by separating each chip with a vertical bar '**|**'.

Example:

```
cpu {
     bus address_bus {
          mau 8;
          mem addr=0 chips=rom_chip; // ref
          ...
     }
     chips rom_chip  attr=r mau=8 size=0x100;  // def
     chips ram_chip  attr=w mau=8 size=0x100;
     ...
}
```

cluster

Syntax:

(Software part)

cluster *cluster_name* **{** *cluster_description* **}**
cluster *cluster_name*[**,** *cluster_name*]... **;**

Description:

In the software layout part you can define the cluster name and cluster location order. The attributes as valid for clusters (see **attribute**) can be specified in the first syntax. If you do not specify any attribute, the default attribute **r** or **w** is automatically set.

In a cluster description you can not only determine the locate order of sections within the named cluster, but you can also specify stack and heap size, extra process memory, define labels for the process, etc.

Example:

```
space address_space {
      block rom {
            cluster first_code_clstr {
             // The default attribute 'r' of cluster
             // text is overruled to  'i'. All
             // sections with attribute 'i' are
             // located here by default.
                  attribute i;
                  amode near_code;
                  amode far_code;
                   // Sections with addressing mode
                   // near_code or far_cdoe are
                   // located here
            }

      block ram {
            cluster data_clstr {
             // default attribute 'w' because the
             // memory is RAM. All writable
             // sections are located here by default.
                  attribute w; // can be omitted
                  amode near_data {
                        section selection=w;
                  }
      }
}
```

contiguous

Syntax:

contiguous { *section_blocks* } (Software part)
contiguous addr = *address* { *section_blocks* }

Description:

A **contiguous** block specifies that all sections named in the block should be located contiguously and in the specified order. The sections will be located back–to–back, only allowing alignment constraint to cause gaps between two consecutive sections.

Alignment constraints of sections are propagated to the contiguous block. This means, for instance, that if the first section should be aligned at even addresses, the contiguous block will start at an even address.

Page constraints will also be propagated to the contiguous block. This means that if a section in the contiguous block may not be located across a specific boundary, the whole contiguous block will not be located over the specified boundaries.

Sections in a contiguous block cannot have conflicting memory requirements, e.g. writable and read–only.

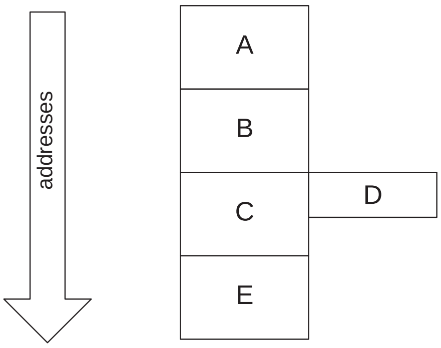
Contiguous blocks can be nested with **overlay** blocks. A block can have an address (**addr=**). Addresses within a block are considered offsets relative to the start of the block.

Example:

The following example shows a contiguous block with a nested overlay block:

```
cluster data_clstr {
    contiguous {
        section A;
        section B;
        overlay {
            section C;
            section D;
        }
        section E;
    }
    ..
}
```

The graphical representation is:



copy

Syntax:

copy *section_name* [**attr =** *attribute*] **;** (Software part)
copy **selection =** *attribute* [**attr =** *attribute*] **;**
copy **;**

Description:

The ROM copy of data sections with the attribute **i** will be copied from ROM to RAM at program startup. With **copy** you define the placement in memory of these ROM copies. You can specify a specific section by giving the section's name, or select sections with a specific attribute. If you do not specify an argument, the locator locates all ROM copies at the specified location. With **attr=** you can change the section attributes.

If you do not specify the keyword **copy** at all, the locator finds a suitable place for ROM copies.

See also the keywords **attribute** and **selection**.

Example:

```
space address_space {
  block rom {
      ...
      cluster code_clstr {
            attribute r; //cluster attribute
            amode far_code {
              table;
              section selection=x;
              section selection=r;
              copy; // all ROM copies are located here
            }
      }
}
```

cpu

Syntax:

cpu { *cpu_description* } (Cpu part)
cpu *filename*

Description:

The keyword **cpu** appears together with **software** and **memory** at the highest level in a description file. The actual cpu description starts between the curly braces **{}**. Normally you do not need to change the cpu part because it is delivered with the product and describes the derivative completely.

The second syntax is the so–called include syntax. The locator opens the file *filename* and reads the actual cpu description from this file. You must start the included file with **cpu** again. The *filename* can contain a complete path including a drive letter (Windows). Parts of *filename*, or the complete *filename* can be put in a environment variable. The file is first searched for in the current directory, and secondly in the `etc` directory relative to the installation directory.

Example:

Contents of the description file:

```
software {
     ...
}

cpu target.cpu //cpu part in separate file
memory target.mem
```

See section 3 for a sample contents of a `.cpu` file.

dst

Syntax:

dst = *address* (Cpu or memory part)

Description:

Specify destination address as part of the keyword **map** in an **amode**, **space** or **bus** description. For *address* you can use any decimal, hexadecimal or octal number. You can also use the (standard) Delfee suffix **k**, for kilo (2^{10}) or **M**, for mega (2^{20}). The unit of measure depends on the MAU (minimum addressable unit) of the destination memory space.

Example:

```
cpu {
  ...
  amode near_code {
      attribute Y1;
      mau 8; // 8-bit addressable
      map src=0 size=1k dst=0 amode=far_code;
  }
}
```

fixed

Syntax:

fixed address = *address* **;** (Software part)
fixed addr = *address* **;** (abbreviated form)

Description:

Define a fixed point in the memory map. The locator allocates the section/cluster preceding the fixed definition and the section/cluster following it as close as possible to the fixed point.

Example:

```
block ram {
     cluster near_data_clstr {
       amode near_data {
          section selection=w;
          fixed  addr = 0x2000;
       }
     }
     cluster far_data_clstr;
}
```

Cluster `far_data_clstr` will be located with its upper bound at address `0x2000` and cluster `near_data_clstr` starts at this address. The same can be applied to sections.

gap

Syntax:

gap; (Software part)
gap length = *value* **;**

Description

Reserve a gap with a dynamic size. The locator tries to make the memory space as big as possible. You can use this keyword in a block description to create a gap between clusters, or in a cluster description to create a gap between sections. You can also use the **gap** keyword in combination with the **fixed** keyword.

With the second form you can specify a gap of a fixed length. This form can only occur in a block description.

Example:

```
space address_space {
     block ram {
          cluster data_clstr {
               attr w;
               amode near_data;
          } // low side mapping

          gap; // balloon
          cluster stck; // high side mapping
     }
}
```

heap

Syntax:

heap *heap_description* **;** (Software part)
heap ;

Description:

Like **table** and **stack**, **heap** is another special section. The section is not created from the `.out` file, but generated at locate time. To control the size of this special section the keyword **length** is allowed within the heap description. You can use **heap** to include dynamic memory for a process.

Heap can only be used if a malloc() function has been implemented.

Two locator labels are used to mark begin and end of the heap, `__lc_bh` for the begin of heap, and `__lc_eh` for the end of heap.

Note that if the **heap** keyword is specified in the description file this does not automatically mean that a heap will always be generated. A heap will only be allocated when its section labels (`__lc_bh` for begin of heap and `__lc_eh` for end of heap) are used in the program.

The heap description can be a length specification and/or an attribute specification. See the example.

Example:

```
layout {
     space address_space {
          block ram {
               cluster data_clstr {
                amode far_data {
                    section selection=w;
                    heap length=100;
                    // Heap of 100 MAUs
                }
               }
          }
     }
}
```

label

Syntax:

label *identifier* ; (Software part)
label = *identifier* ; (All parts)

Description:

The first form can be used stand–alone to specify a virtual address in memory by means of a label. The virtual address is label **__lc_u_***identifier*. Note that at C level, all locator labels start with one underscore (the compiler adds another underscore '_').

The second form can only be used as part of another keyword. As part of the keyword **reserved** you can assign a label to an address range. The start of the address range is identified by label **__lc_ub_***identifier*. The end of the address range is identified by label **__lc_ue_***identifier*. The keyword **label** is also allowed as part of the **map** keyword to assign a name to a block of memory in a space definition.

Example:

From the software part:

```
block ram {
     cluster data_clstr {
          attribute w;
          amode far_data {
               section selection=w;
               heap;
               stack;
               reserved label=xvwbuffer length=0x10;

               // Start address of reserved area is
               // label __lc_ub_xvwbuffer
               // End address of reserved area is
               // label __lc_ue_xvwbuffer
          }
     }
}
```

From the cpu part:

```
space address_space {
      mau 8;
      map src=0   size=32k dst=0   bus = address_bus label=rom;
      map src=32k size=32k dst=32k bus = address_bus label=ram;
}
```

layout

Syntax:

layout { *layout_description* } (Software part)
layout *filename*

Description:

The **layout** part describes the layout of sections in memory. The **layout** part groups sections into clusters and you can define the name, number and the order of clusters. The **layout** part describes how these clusters must be allocated into physical RAM and ROM block. The space and block names used in the **layout** part must be present in the memory part or the cpu part. The cluster definitions can contain fixed addresses as well as definitions of gaps between sections.

Example:

```
software {
  layout {
    space address_space {
     block rom {
       cluster first_code_clstr {
         attribute i;
         amode near_code;
       }
     ....
```

length

Syntax:

length = *length* (Cpu, memory and software part)
leng = *length* (abbreviated form)

Description:

You can use the keyword **length** to define the length in MAUs (minimum addressable units) of a certain memory area. *length* must be a numeric value and can be given either in hex, octal or decimal. As usual, hex numbers must start with '0x' and octal numbers must start with '0'. You can use the suffix **k** which stands for kilo or **M** which stands for mega.

You can use **length** to specify the length of the reserved memory or to specify the stack, heap or gap length. For details see the keywords **reserved**, **stack**, **heap** and **gap**.

Example:

```
space address_space {
  block ram {
     cluster data_clstr {
        amode far_data {
          stack leng = 2k;
        }
     }
  }
}
```

load_mod

Syntax:

load_mod *identifier* **start =** *label*; (Software part)
load_mod start = *label*;

Description:

With **load_mod** you are introducing a load module description. This keyword is followed by an optional identifier, representing a load module name with or without the `.out` extension. The load module itself must be supplied to the locator as a parameter in the invocation. If the identifier is omitted, the load module is taken from the command line.

Example:

```
software {
     load_mod start = __START;
}
```

or:

```
software {
     load_mod hello start = __USER_start;
}
```

map

Syntax:

map *map_description* (Cpu or memory part)

Description:

Map a memory part, specified as a source address and a size, to a destination address of an **amode**, **space** or **bus**. The unit of measure depends on the MAU of the memory space.

Example:

```
cpu {
  .
  amode far_data {
       attribute Y4;
       mau 8;
       map src=0 size=32k dst=32k space=address_space;
  }
  space address_space {
       mau 8;
       map src=0   size=32k dst=0   bus = address_bus label=rom;
       map src=32k size=32k dst=32k bus = address_bus label=ram;
  }
  bus address_bus {
       mau 8;
       mem addr=0   chips=rom_chip;
       map src=0x100 size=0x7f00  dst=0x100 bus=external_rom_bus;
       mem addr=32k chips=ram_chip;
       map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
  }
  .
}
```

mau

Syntax:

mau *number* **;** (Cpu or memory part)
mau = *number*

Description:

You can use the keyword **mau** to specify the minimum addressable unit in bits of a certain memory area. The first form can only be used in an **amode**, **space** or **bus** description. The second form can be used to specify the minimum addressable unit of a chip. Note that **mau** affects the unit of measure for other keywords. If no **mau** is specified, the default number is 8 (byte addressable).

Example:

```
cpu {
  amode near_code {
     attribute Y1;
     mau 8;  // byte addressable
     map src=0 size=1k dst=0 amode=far_code;
     // src is at address 0,
     // size is 1k byte units
     // dst is at address 0
  }
}
```

mem

Syntax:

mem *mem_description* ; (Cpu or memory part)

Description:

Define the start address of a chip in memory. The only keywords allowed in a mem description are **address** and **chips**.

Example:

```
cpu {
  ...
  bus internal_bus {
    mau 8;
    mem addr=0    chips=rom_chip;
     // chip ’rom_chip’ is located at memory
     // address 0
     ...
    mem addr=32k  chips=ram_chip;
     // chip ’ram_chip’ is located at memory
     // address 0x8000
     ...

  }
  chips rom_chip  attr=r mau=8 size=0x100;
  chips ram_chip  attr=w mau=8 size=0x100;

}
```

memory

Syntax:

memory { *memory_description* } (Memory part)
memory *filename*

Description:

Together with **software** and **cpu**, **memory** introduces a main part of the description file. You can specify the actual memory part between the curly braces **{}**.

You can use the memory part to describe any additional memory or addresses of peripherals not integrated on the cpu.

The second syntax is the include syntax. In this case, the memory part is defined in a separate file. This included file must start again with **memory**. The *filename* can contain a complete path, including a drive letter (Windows). You can put parts of *filename*, or the complete *filename* in an environment variable. The file is first searched for in the current directory, and secondly in the `etc` directory relative to the installation directory.

Example:

```
software {
     ...
}

cpu target.cpu
memory target.mem    //mem part in separate file
```

See section 5 for a sample contents of a `.mem` file.

overlay

Syntax:

overlay { *section_blocks* } (Software part)
overlay addr = *address* { *section_blocks* }

Description:

An **overlay** block specifies that all sections named in the block should start at the same address.

Page constraints will also be propagated to the overlay block. This means that if a section in the overlay block may not be located across a specific boundary, the whole overlay block will not be located over the specified boundaries.

Sections in an overlay block cannot have conflicting memory requirements, e.g. writable and read–only.

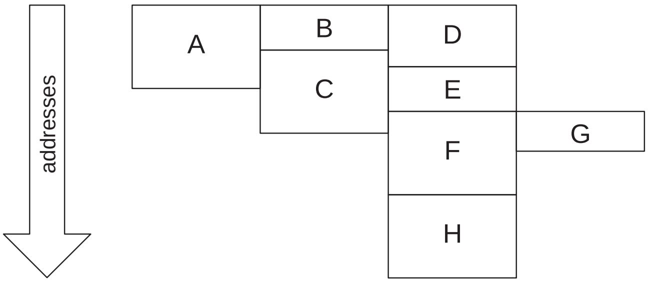
Overlay blocks can be nested with **contiguous** blocks. Overlay blocks can have an absolute address specified (**addr=**).

Example:

The following example shows an overlay block contiguous block with a nested overlay block:

```
cluster data_clstr {
    overlay {
        section A;
        contiguous {
            section B;
            section C;
        }
        contiguous {
            section D;
            section E;
            overlay {
                section F;
                section G;
            }
            section H;
        }
    }
    ..
}
```

The graphical representation is:



regsfr

Syntax:

regsfr *filename* (Cpu or memory part)

Description:

Specify a register file generated by the register manager for use by the CrossView debugger.

Example:

```
cpu {
     .
     .
     .
     regsfr regfile.dat
     /*
      * Use file regfile.dat generated by
      * register manager for CrossView
      */
}
```

reserved

Syntax:

reserved *reserved_description* **;** (Software part)
reserved;

Description:

Reserve a fixed amount of memory space or reserve as much memory as possible in the memory space. If no length is specified the size of the memory allocation depends on the size of the memory space or the size is limited by a fixed point definition following the **reserved** allocation.

You can only use the keywords **address**, **attribute**, **label** and **length** in the reserved description. You can use the keyword **reserved** in an amode description.

Example:

```
space address_space {
     block rom {
          cluster code_clstr {
               amode near_code {
                 // system reserved
                 // (exception vector)
                 reserved length=0x2 addr=0x24;
               }
     }
}
```

section

Syntax:

(Software part)

section *identifier* [**addr =** *address*] [**attr =** *attribute*] **;**
section **selection =** *attribute* [**addr =** *address*] [**attr =** *attribute*]**;**

Description:

section can be used in the layout part to specify the location order within a cluster. See also **layout**.

The *identifier* is the name of a section.

With **addr=** you can make a section absolute.

With **attr=** you can assign new attributes to a section or disable attributes.

See also the keywords **address**, **attribute** and **selection**.

Example:

```
space address_space {
     block ram {
          cluster data_clstr {
               amode near_data {
                 // locate section .data here and set
                 // attribute 'w'
                 section .data attr=w;
                 section selection=b attr=-b;
               }
          }
     }
}
```

selection

Syntax:

selection = *attribute*

Description:

You can use **selection** after the keywords **section** or **copy** to select all sections with (a) specified attribute(s).

If more attributes are specified, only sections with all attributes are selected. If a minus sign '**–**' precedes the attribute, only sections **not** having the attribute are selected.

See also the keywords **attribute**, **copy** and **section**.

Example:

```
space address_space {
     block ram {
       cluster data_clstr {
          amode near_data {
           // select sections with w on and not i.
           // (select all writable sections which
           // are not copied from ROM)
           section selection=-iw;
          }
       }
     }
     .
}
...
```

size

Syntax:

size = *size* (Cpu or memory part)

Description:

You can use the keyword **size** to define the size in minimum addressable units (MAU) of a certain memory area. *size* must be a numeric value and can be given either in hex, octal or decimal. As usual, hex numbers must start with '0x' and octal numbers must start with '0'. You can use the suffix **k** which stands for kilo or **M** which stands for mega.

You can use **size** to specify the size of a part of memory that must be mapped on another part of memory or to specify the the size of a chip. For details see the keywords **map** and **chips**.

Example:

```
cpu {
  amode near_code {
      attribute Y1;  //identify near_code with Y1
      map src=0 size=1k dst=0 amode=far_code;
  }
space address_space {
      mau 8;
      map src=0   size=32k dst=0   bus=address_bus label=rom;
      map src=32k size=32k dst=32k bus=address_bus label=ram;
}
  chips rom_chip attr=r mau=8 size=0x100;
  chips ram_chip attr=w mau=8 size=0x100;
      // size of chips
}
```

software

Syntax:

software { *software_description* } (Software part)
software *filename*

Description:

The keyword **software** appears at the highest level in a description file. The actual software description starts between the curly braces **{}**.

The second syntax is the so called include syntax. The locator will open file *filename* and read the actual software description from this file. The first keyword in *filename* must be **software** again. The *filename* can contain a complete path including a drive letter (Windows). You can put parts of *filename*, or the complete *filename* in an environment variable. The file is first searched for in the current directory, and secondly in the `etc` directory relative to the installation directory.

Example:

Contents of the description file:

```
software $(MY_OWN_DESCRIPTION)

cpu target.cpu
memory target.mem
```

Environment variable `MY_OWN_DESCRIPTION` contains the name of a file with contents like:

```
software {
     load_mod start = __START;
     layout {
     .
     .
     .
     }
}
```

space

Syntax:

space *identifier* **{** *space_description* **}** (Software part)

(Cpu or memory part)

space *identifier*[**,** *identifier*]... **{** *space_description* **}**
space = *identifier*

Description:

The keyword **space** can be used in the cpu part, memory part and software part. In the cpu or memory part you can use **space** to describe a physical memory address space. The only keywords allowed in a space description in the cpu or memory part are **mau** and **map**.

In the software part you can use **space** to describe one or more memory blocks. Each space has a symbolic name as previously defined by the keyword **space** in the cpu or memory part.

Example:

From the cpu part:

```
cpu {
  amode far_data {
      attribute Y4;
      mau 8;
      map src=0 size=32k dst=32k space=address_space;
  }
      ...

  space address_space {
      // Specify space 'address_space' for the
      // address_bus address bus.
      mau 8;
      map src=0   size=32k dst=0   bus=address_bus label=rom;
      map src=32k size=32k dst=32k bus=address_bus label=ram;
  }
  .
}
```

From the software part:

```
layout {
      // define the preferred locating order of sections
      // in the memory space
      // (the range is defined in the .cpu file)
      space address_space {
      ...

            // define for each sub-area in the space
            // the locating order of sections
            block rom {
              // Memory block starting at chip rom_chip

              // define a cluster for read-only sections
              cluster code_clstr {
                  ....
              }
            }
      .
      }
}
```

src

Syntax:

src = *address* (Cpu or memory part)

Description:

Specify source address as part of the keyword **map** in an **amode**, **space** or **bus** description. For *address* you can use any decimal, hexadecimal or octal number. You can also use the (standard) Delfee suffix **k**, for kilo (2^{10}) or **M**, for mega (2^{20}). The address is specified in the addressing mode's local MAU (minimum addressable unit) size (default 8 bits).

Example:

```
cpu {
  ...
  amode near_code {
      attribute Y1;
      mau 8; // 8-bit addressable
      map src=0 size=1k dst=0 amode=far_code;
  }
}
```

stack

Syntax:

stack *stack_description* ; (Software part)
stack ;

Description:

stack is a special form of a section description. The stack is allocated at locate time. The locator only allocates a stack if one is needed. Two special locator labels are associated with the stack space located with keyword **stack**. The begin of the stack area can be obtained by the locator label `__lc_bs`, the end address is accessible by means of label `__lc_es`.

You can only use the keywords **attribute** and **length** in the stack description. If you specify **stack** without a description, the locator tries to make the stack as big as possible. If you do not specify the keyword **stack** at all, the locator also tries to make the stack as big as possible but at least 100 (MAUs).

Example:

```
space address_space {
     block ram {
          cluster data_clstr {
               amode far_data {
                    section selection=w;
                    stack leng=150;
                    // stack of 150 MAUs
                    ...
               }
          }
     }
}
```

start

Syntax:

start = *label* **;** (Software part)

Description:

Define a start label for a process.

You can use **start** only within a load module description.

Example:

```
software {
     load_mod start = system_start;

     layout {
     .
     .
     }
}
```

table

Syntax:

table attr = *attribute* **;** (Software part)
table ;

Description:

Like **stack** and **heap** also **table** is a special kind of section. Normal sections are generated at compile time, and passed via the assembler and linker to the locator. The stack and heap sections are generated at locate time, with a user requested size.

table is different. The locator is able to generate a copy table. Normally, this table is put in read–only memory. If you want to steer the table location, you can use the **table** keyword. With table only **attribute** is allowed. The length is calculated at locate time. **table** can occur in a cluster description.

Example:

```
space address_space {
  block rom {
      ...
      cluster code_clstr {
           attribute r; // cluster attribute
           amode far_code {
             table; // locate copy table here
             section selection=x;
             section selection=r;
             copy; // all ROM copies are located here
           }
      }
}
```

7.1 ABBREVIATION OF DELFEE KEYWORDS

The following Delfee keywords can be abbreviated to unique 4 character words:

Keyword	Abbreviation
address	addr
assert	asse
attribute	attr
length	leng

Table G–3: Abbreviation of Delfee keywords

7.2 DELFEE KEYWORDS SUMMARY

Keyword	Description
address	Specify absolute memory address
align	Specify section alignment
amode	Specify the addressing modes
assert	Error if assertion failed
attribute	Assign attributes to clusters, sections, stack or heap
block	Define physical memory area
bus	Specify address bus
chips	Specify cpu chips
cluster	Specify the order and placement of clusters
contiguous	Specify a contiguous block of sections
copy	Define placement of ROM–copies of data sections
cpu	Define cpu part
dst	Destination address
fixed	Define fixed point in memory map
gap	Reserve dynamic memory gap
heap	Define heap
label	Define virtual address label
layout	Start of the layout description

Keyword	Description
length	Length of stack, heap, physical block or reserved space
load_mod	Define load module (process)
map	Map a source address on a destination address
mau	Define minimum addressable unit (in bits)
mem	Define physical start address of a chip
memory	Define memory part
overlay	Specify a block of sections which must be overlaid
regsfr	Specify register file for use by CrossView
reserved	Reserve memory
section	Define how a section must be located
selection	Specify attributes for grouping sections into clusters
size	Size of address space or memory
software	Define the software part
space	Define an addressing space or specify memory blocks
src	Source address
stack	Define a stack section
start	Give an alternative start label
table	Define a table section

Table G–4: Overview of Delfee keywords

APPENDIX H

DELFEE SYNTAX

TASKING

APPENDIX H

This appendix describes the Delfee description language.

GENERAL

description
    partition
    description partition

partition
    memory_partition
    cpu_partition
    software_partition

ident_list
    ident_list **,** *identifier*
    identifier

identifier
    STRING

file_name
    STRING

CPU

cpu_partition
    cpu { *static_specs_list* **}**
    cpu { }
    cpu *file_name*

MEMORY

memory_partition
    memory { *static_specs_list* **}**
    memory { }
    memory *file_name*

static_specs_list
    static_specs_list static_specs
    static_specs

static_specs
  amod_specs
  spce_specs
  bus_specs
  chips_specs

amod_specs
  amode *ident_list* { *amod_list* }

spce_specs
  space *ident_list* { *spce_list* }

bus_specs
  bus *ident_list* { *bus_list* }

chips_specs
  chips *ident_list chips_list* **;**

amod_list
  amod_list amod_def
  amod_def

spce_list
  spce_list spce_def
  spce_def

bus_list
  bus_list bus_def
  bus_def

chips_list
  chips_list chips_def
  chips_def

amod_def
  mau_spec
  attribute_spec
  map_spec

spce_def
  mau_spec
  map_spec

bus_def
  mau_spec
  mem_spec
  map_spec

chips_def
  mau_equ_spec
  attribute_equ_spec
  size_spec

mau_spec
  mau *NUMBER* **;**

mau_equ_spec
  mau = *NUMBER*

attribute_spec
  attribute *STRING* **;**
  attribute *NUMBER* **;**
  attr *STRING* **;**
  attr *NUMBER* **;**

attribute_equ_spec
  attribute = *STRING*
  attribute = *NUMBER*
  attr = *STRING*
  attr = *NUMBER*

map_spec
  map *map_list* **;**

map_list
  map_list map_def
  map_def

map_def
  src_spec
  size_spec
  dst_spec
  align_spec
  page_spec
  amode_spec
  space_spec
  bus_spec

mem_spec
  mem *mem_list* **;**

mem_list
  mem_list mem_def
  mem_def

mem_def
  addr_spec
  chips_spec

src_spec
  src = *NUMBER*

size_spec
  size = *NUMBER*

dst_spec
  dst = *NUMBER*

align_spec
  align = *NUMBER*

page_spec
  page = *NUMBER*

amode_spec
  amode = *identifier*

space_spec
  space = *identifier*

bus_spec
  bus = *low_bus_list*

addr_spec
  address = *NUMBER*
  addr = *NUMBER*

chips_spec
  chips = *low_chip_list*

low_bus_list
  low_bus_list **|** *identifier*
  identifier

low_chip_list
    low_chip_list **,** *low_chip_pair*
    low_chip_pair

low_chip_pair
    low_chip_pair **|** *low_chip*
    low_chip

low_chip
    identifier

SOFTWARE

software_partition
    software { *layout_blocks* **}**
    software { }
    software *file_name*

layout_blocks
    layout_blocks layout_block
    layout_block

layout_block
    layout
    loadmod

loadmod
    load_mod *software_specs* **;**
    load_mod *identifier software_specs* **;**

software_specs
    software_specs software_spec
    software_spec

software_spec
    start
    process

start
    start = *identifier* **;**

process
    process = *pids*

pids
    NUMBER
    pids **,** *NUMBER*

layout
    layout { *space_blocks* **}**
    layout { }
    layout *file_name*

space_blocks
    space_blocks space_block
    space_block

space_block
    space *identifier* **{** *block_blocks* **}**

block_blocks
    block_blocks block_block
    block_block

block_block
    block *identifier* **{** *cluster_blocks* **}**

cluster_blocks
    cluster_blocks cluster_block
    cluster_block

cluster_block
    cluster_spec
    p_gap_spec
    p_fixed_spec
    p_pool_spec
    p_skip_spec
    p_label_spec

cluster_spec
    cluster *identifier* **{** *amod_blocks* **}**
    cluster *ident_list* **;**

amode_blocks
    amode_blocks amode_block
    amode_block

amode_block
  amode *ident_list* { *section_blocks* }
  amode *ident_list* **;**
  section_block

p_gap_spec
  gap *length* **;**
  gap ;

p_fixed_spec
  fixed *address* **;**

p_pool_spec
  pool *length* **;**
  pool ;

p_label_spec
  label *identifier* **;**

p_skip_spec
  skip ;

attribute
  attribute_equ_spec

length
  length = *NUMBER*
  leng = *NUMBER*

address
  address = *NUMBER*
  addr = *NUMBER*

section_blocks
  section_blocks section_block
  section_block

section_block
  section_spec
  copy_spec
  v_fixed_spec
  v_gap_spec
  v_reserved_spec
  stack_spec
  heap_spec
  table_spec
  others
  label_spec
  v_assert_spec
  attribute_spec
  contiguous_block
  overlay_block

contiguous_block
  contiguous *address* { *virtual_block_entries* }
  contiguous { *virtual_block_entries* }

overlay_block
  overlay *address* { *virtual_block_entries* }
  overlay { *virtual_block_entries* }

virtual_block_entries
  virtual_block_entries virtual_block_entry
  virtual_block_entry

virtual_block_entry
  section_spec
  copy_spec
  label_spec
  contiguous_block
  overlay_block

section_spec
  section *selection modifiers* **;**
  section *selection* **;**

modifiers
  modifiers modifier
  modifier

modifier
    attribute
    address

copy_spec
    copy *selection attribute* **;**
    copy *selection* **;**
    copy ;

selection
    selection = *STRING*
    identifier

v_fixed_spec
    fixed *address* **;**

v_gap_spec
    gap ;

v_reserved_spec
    reserved *reserved_options* **;**
    reserved ;

reserved_options
    reserved_options reserved_option
    reserved_option

reserved_option
    attribute
    address
    length
    label_equ_spec

stack_spec
    stack *stack_options* **;**
    stack ;

heap_spec
    heap *stack_options* **;**
    heap ;

stack_options
    stack_options stack_option
    stack_option

stack_option
  attribute
  length

table_spec
  table *attribute* **;**
  table ;

others
  others ;

label_spec
  label *identifier* **;**

label_equ_spec
  label = *identifier*

v_assert_spec
  assert (*bool_expression* **,** *STRING* **) ;**
  asse (*bool_expression* **,** *STRING* **) ;**

bool_expression
  termp bool_op termp

termp
  term **+** *termp*
  term **–** *termp*
  term

term
  (*term* **)**
  identifier
  NUMBER

bool_op
  <
  >
  ==
  !=

A **NUMBER** is a series of (hex) digits with optional suffixes ’k’ ’M’ ’G’ which stands for ’kilo’, ’mega’ and ’giga’. Numbers may be given in hex, octal or decimal with the usual prefix. Where applicable numbers may be preceded by a minus sign.

A **STRING** is a series of characters that is not a number (089 is a **STRING** because it is not a valid octal number) and consists of alphanumeric characters including '_', '.', '–' and the host dependent directory separators. (For PC '\', '/' and ':')

Any (part of a) token may contain environment variables. If the environment variable A contains the text 'foo' then the sequence:

```
$A/proto.dsc
```

is translated to:

```
foo/proto.dsc
```

Multi character variables must be combined with braces:

```
window = $(MODE);
```

There are three methods to write comments in a delfee script. The first one is the 'C' style comment between '/*' and '*/'. The second form is a '#' in the first column. The second form allows preprocessing by the C–preprocessor. Any *#line* or *#file* directive will be ignored by the locator. The third form is the 'C++' style comment; a double slash '//' anywhere on a line introduces comments until the end of line.

APPENDIX I

IEEE–695 OBJECT FORMAT

TASKING

APPENDIX

1 TIOF AND IEEE-695

The IEEE–695 standard describes MUFOM: Microprocessor Universal Format for Object Modules. It defines a target independent storage standard for object files. However, this standard does not describe how symbolic debug information should be encoded according to that standard. Symbolic debug information can be a part of an object file. A debugger which reads an object file uses the symbolic debug information to obtain knowledge about the relation between the executable code and the origination high–level language source files. Since the IEEE–695 standard does not describe the representation of debug information, working implementations of this standard show vendor specific and microprocessor specific solutions for this area.

TIOF, which stands for Target Independent Object Format, is specified as a MUFOM based standard including the representation of symbolic debug information for high–level languages, without introducing the microprocessor dependent solutions. The current version of the TASKING debugger is not yet prepared to read TIOF, so you will have to select IEEE–695 as output format of the locator when you want to debug a program.

Since TIOF and IEEE–695 both use the MUFOM concept as their basis both formats are very similar.

2 COMMAND LANGUAGE CONCEPT

Most object formats are record oriented: there are one or more section headers at a fixed position in the file which describe how many sections are present. A section header contains information like start address, file offset, etc. The contents of the section is in some data part, which can only be processed after the header has been read. So the tool that reads such an object uses implicit assumptions how to process such a file. Seeking through the file to get those records which are relevant is usual.

MUFOM (IEEE–695) uses a different approach. It is designed as a command language which steers the linker, locator and object reader in the debugger.

An assembler or compiler may create an object module where most of the data contained in it is relocatable. The next phase in the translation process is linking several object modules into one new object module. A relocatable object uses relocation expressions at places where the absolute values are not yet known. An expression evaluator in the locator transforms the relocation expressions into absolute values.

Finally the object is ready for loading into memory. Since an object file is transformed by several processes, MUFOM implements an object file as a sequence of commands which steers this transformation process.

These commands are created, executed or copied by one of five processes which act on a MUFOM object file:

1. Creation process
   Creation of the object file by an assembler or compiler. The assembler or compiler tells other MUFOM processes what to do, by emitting commands generated from assembly source text or a high–level language.

2. Linkage process
   Linking of several object modules into one module resolving external references by renaming X variables into I variables, and by generating new commands (assigning of R variables).

3. Relocation process
   Relocation, giving all sections an absolute address by assigning their L variable.

4. Expression evaluation process
   Evaluation of loader expressions, generated in one of the three previously mentioned MUFOM processes.

5. Loader process
   Loading the absolute memory image.

The last four processes are in fact command interpreters: the assembler writes an object file which is basically a large sequence of instructions for the linker. For example, instead of writing the contents of a section as a sequence of bytes at a specific position in the file, IEEE–695 defines a load command, LR, which instructs the linker to load a number of bytes. The LR command specifies the number of MAUs (minimum addressable unit) that will be relocated, followed by the actual data. This data can be a number of absolute bytes, or an expression which must be evaluated by the linker.

Transforming relocation expressions into new expressions or absolute data and combining sections is the actual linkage process.

It is possible that one or more of the above MUFOM processes are combined in one tool. For instance, the locator is built from process 3 and process 4 above.

3 NOTATIONAL CONVENTIONS

The following conventions are used in this appendix:

|	select one of the items listed between '|'
" "	literal characters are between " "
[ ]+	optional item repeats one time or more
[ ]?	optional item repeats zero times or one time
[ ]*	optional item repeats zero times or more
::=	can be read as "is defined as"

4 EXPRESSIONS

An expression in an IEEE–695 file or a TIOF file is a combination of variables, operators and absolute data.

The variable name always starts with a non–hexadecimal letter (G...Z), immediately followed by an optional hexadecimal number. The first non–hexadecimal letter gives the class of the variable. Reading an object file you encounter the following variables:

G – Start address of a program. If not assigned this address defaults to the address of low–level symbol **_start**.

I – An I variable represents a global symbol in an object module.

The I variable is assigned an expression which is to be made available to other modules for the purpose of linkage edition. The name of an I variable is always composed of the letter 'I', followed by a hexadecimal number. An I variable is created only by an NI command.

L –	Start address of a section. This variable is only used for absolute sections. The 'L' is followed by a section index, which is an hexadecimal number. L variables are created by an assignment command, but the section index must have been been defined by an ST command.
N –	Name of internal symbol. This variable is used to assign values of local symbols, or, to build complex types for use by a high–level language debugger, or for inter–modular type checking during linkage. The N variable is created with a NN command.
P –	Program pointer per section. This variable always contains the current address of the target memory location. The P variable is followed by a section index, which is a hexadecimal number. The section index must have been defined with an ST command (section type command). The variable is created after its first assignment.
R –	The R type variable is a relocation reference for a particular section. All references to addresses in this section must be made relative to the R variable. Linking is accomplished by assigning a new value to R. The R variable consists of the letter 'R', followed by an section index, which is a hexadecimal number. The section index must have been defined with an ST command. The default value of an (unassigned) R variable is 0.
S –	The S type variable is the section size (in MAUs) for a section. There is one S variable per section. The 'S' is followed by an section index. An S variable is created by its first assignment.
W –	Work variable. This type of variable can be used to assign values to, which can be used in following MUFOM commands. They serve the purpose of maintaining values in a workspace without any additional meaning. A work variable consists of the letter 'W' followed by a hexadecimal number. W variables are created by their first assignment.
X –	An X type variable refers to an external reference. X–variables cannot have a value assigned to it. An X variable consists of the letter 'X' followed by a hexadecimal number.

The MUFOM language uses the following data types to form expressions:

digit	::=	”0” | ”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”
hex_letter	::=	”A” | ”B” | ”C” | ”D” | ”E” | ”F”
hex_digit	::=	digit | hex_letter
hex_number	::=	[hex_digit]+
nonhex_letter	::=	”G” | ”H” | ”I” | ”J” | ”K” | ”L” | ”M” | ”N” | ”O” | ”P” | ”Q” | ”R” | ”S” | ”T” | ”U” | ”V” | ”W” | ”X” | ”Y” | ”Z”
letter	::=	hex_letter | nonhex_letter
alpha_num	::=	letter | digit
identifier	::=	letter [alpha_num]*
character	::=	’value valid within chosen character set’
char_string_length	::=	hex_digit hex_digit
char_string	::=	char_string_length [character]*

The numeric value specified in ’char_string_length’ should be followed by an equal number of characters.

Expressions may be formed out of immediate numbers and MUFOM variables. The MUFOM processes 2 to 4, which form the linker and the locator, contain expression evaluators which parse and calculate the values for the expressions. If a MUFOM process cannot calculate the absolute value of an expression, because the values of the variable are not yet known, it copies the expression (with modifications) into the output file.

Expression are coded in reverse Polish notation. (The operator follows the operands.)

```
expression ::= boolean_function |
               one_operand_function |
               two_operand_function |
               three_operand_function |
               four_operand_function |
               conditional_expr | hex_number | MUFOM_variable
```

4.1 FUNCTIONS WITHOUT OPERANDS

@F : false function

@T : true function

```
boolean_function ::= ”@F” | ”@T”
```

The false and true function produce a boolean result false or true which may be used in logical expressions. Both functions do not have operands.

4.2 MONADIC FUNCTIONS

Monadic functions have one operand which precedes the function.

```
one_operand_function ::= operand ”,” monop
operand   ::= expression
monop     ::= ”@ABS” | ”@NEG” | ”@NOT” | ”@ISDEF”
```

@ABS:	returns the absolute value of an integer operand
@NEG:	returns the negative value of an integer operand
@NOT:	returns the negation of a boolean operand or the one’s complement value if the operand is an integer
@ISDEF:	returns the logical true value if all variable in an expression are defined, return false otherwise.

4.3 DYADIC FUNCTIONS AND OPERATORS

Dyadic functions and operators have two operands which precede the operator or function.

```
two_operand_function ::= operand1 ”,” operand2 ”,” dyadop

operand1 ::= expression

operand2 ::= expression

dyadop       ::= ”@AND” | ”@MAX” | ”@MIN” | ”@MOD” |
                 ”@OR” |”@XOR” |
                 ”+” | ”–” | ”/” | ”*” | ”<” | ”>” | ”=” | ”#”
```

@AND:	returns boolean true/false result of logical ’and’ operation on operands, when both operands are logical values. When both operands are not logical values the bitwise and is performed.
@MAX:	compares both operands arithmetically and returns the largest value.
@MIN:	compares both operands arithmetically and returns the smallest value.
@MOD:	returns the modulo result of the division of operand1 by operand2. The result is undefined if either operand is negative, or if operand2 is zero.
@OR:	returns boolean true/false result of logical ’or’ operation on operands, when both operands are logical values. When both operands are no logical values the bitwise and is performed.
+, −, *, /:	These are the arithmetic operators for addition, subtraction, multiplication and division. The result is an integer. For division the result is undefined if operand2 equals zero. The result of a division rounds toward zero.
<, >, =, #:	These are operators for the following logical relations: ’less than’, ’greater than’, ’equals’, ’is unequal’. The result is true or false.

4.4 MUFOM VARIABLES

The meaning of the MUFOM variable is explained in section 4. The following syntax rules apply for the MUFOM variables:

```
MUFOM_variable        ::=     MUFOM_var |
                              MUFOM_var_num
                              MUFOM_var_optnum

MUFOM_var             ::=     ”G”

MUFOM_var_num         ::=     ”I” | ”N” | ”W” | ”X”
                              hex_number

MUFOM_var_optnum      ::=     ”L” | ”P” | ”R” | ”S”
                              [ hex_number ]?
```

4.5 @INS AND @EXT OPERATOR

The @INS operator inserts a bit string.

```
four_operand_function  ::=    operand1 ”,” operand2 ”,” operand3
                              ”,” operand4 ”,” @INS
```

operand2 is inserted in operand1 starting at position operand3, and ending at position operand4.

The @EXT operator extracts a bit string.

```
three_operand_function ::=    operand1 ”,” operand2 ”,” operand3
                              ”,” @EXT
```

A bit string is extracted from operand1 starting at position operand2 and ending at position operand3.

4.6 CONDITIONAL EXPRESSIONS

```
conditional_expr  ::=  err_expr | if_else_expr

err_expr          ::=  value ”,” condition ”,” err_num ”,” ”@ERR”

value             ::=  expression

condition         ::=  expression

err_num           ::=  expression

if_else_expr      ::=  condition ”,” ”@IF” ”,” expression ”,”
                       ”@ELSE” ”,” expression ”,” ”@END”
```

5 MUFOM COMMANDS

5.1 MODULE LEVEL COMMANDS

At module level there are four commands: one command to start and one to end a module, one command to set the date and time of creation of the module, and one command to specify address formats.

5.1.1 MB COMMAND

The MB command is the first command in a module. It specifies the target machine configuration and an optional command with the module name.

MB_command ::= ”MB” machine_identifier [”,” module_name]? ”.”

Example: `MB 5600x.`

5.1.2 ME COMMAND

The module end command is the last command in an object file. It defines the end of the object module.

ME_command ::= ”ME.”

5.1.3 DT COMMAND

The DT command sets the date and time of creation of an object module.

DT_command ::= ”DT” [digit]* ”.”

Example: `DT19930120120432.`

The format of display of the date and time is ”YYYYMMDDHHMMSS”:

4 digits for the year, 2 digits for the month, 2 digits for the day, 2 digits for the hour, 2 digits for the minutes and 2 digits for the seconds.

5.1.4 AD COMMAND

The AD command specifies the address format of the target execution environment.

```
AD_command       ::=  ”AD” bits_per_MAU [ ”,” MAU_per_address
                       [ ”,” order ]? ]?

MAU_per_address  ::=  hex_number

bits_per_MAU     ::=  hex_number

order            ::=  ”L” | ”M”
```

MAU stands for minimum addressable unit. This is target processor dependant.

L means least significant byte at lowest address (little endian)
M means most significant byte at lowest address (big endian)

Example:

```
AD24,1,L.
```

Specifies a 1–word addressable 24–bit processor running in little endian mode.

5.2 COMMENT AND CHECKSUM COMMAND

The comment command offers the possibility to store information in an object module about the object module and the translators that created it. The comment may be used to record the file name of the source file of the object module or the version number of the translator that created it. Because the standard supports several layers each of which has its own revision number an object module may contain several comment commands which specify which revision of the standard has been used to create the module. The contents of a comment is not prescribed by the standard and thus it is implementation defined how a MUFOM process handles a comment command.

```
CO_command     ::= ”CO” [comment_level]? ”,” comment_text ”.”

comment_level  ::= hex_number

comment_text   ::= char_string
```

The comment levels 0 – 6 are reserved to pass information about the revision number of the layers in this standard.

The checksum command starts and checks the checksum calculation of an object module.

5.3 SECTIONS

A section is the smallest unit of code or data that can be controlled separately. Each section has a unique number which is introduced at the first section begin (SB) command. The contents of a section may follow its introduction. A section ends at the next SB command with a number different from the current number. A section resumes at an SB command with a number that has been introduced before.

5.3.1 SB COMMAND

```
SB_command    ::=  ”SB” hex_number ”.”
```

The maximum number of sections in an object module is implementation defined.

5.3.2 ST COMMAND

The ST command specifies the type of a section.

```
ST_command      ::=   ”ST” section_number [ ”,” section_type ]*
                                          [ ”,” section_name ]? ”.”

section_type    ::=   letter

section_name    ::=   char_string
```

A section can be named or unnamed. If section_name is omitted a section is unnamed. A section can be relocatable or absolute. If the section start address is an absolute number the section is called absolute. If the section start address is not yet known, the section is called relocatable. In relocatable sections all addresses are specified relative to the relocation base of that section. The relocation phase of the linker or locator may map the relocation base of a section onto a fixed address.

During linkage edition the section name and the section attributes identify a section and thus the actions to be taken. If a section is defined in several modules, the linkage editor must determine how to act on sections with the same name. This can be either one of the following strategies:

- several sections are to be joined into a single one
- several sections are to be overlapped
- sections are not to coexist

A section type gives additional information to the linkage editor about the section, which may be used to layout a section in memory. Section type information is encoded with letters, which may be combined in one ST command. Some combinations of letters are invalid or may be meaningless.

letter	**meaning**	**class**	**explanation**
A	absolute	access	section has absolute address assigned to corresponding L–variable
R	read only	access	no write access to this section
W	writable	access	section may be read and written
X	executable	access	section contains executable code
Z	zero page	access	if target has zero page or short addressable page Z–section map into it
Y*num*	addressing mode	access	section must be located in addressing mode *num*
B	blank	access	section must be initialized to ’0’ (cleared)
F	not filled	access	section is not filled or cleared (scratch)
I	initialize	access	section must be initialized in rom
E	equal	overlap	if sections in two modules have different length an error must be raised
M	max	overlap	Use largest value as section size
U	unique	overlap	The section name must be unique
C	cumulative	overlap	Concatenate sections if they appear in several modules. The section alignment for partial section must be preserved

letter	meaning	class	explanation
O	overlay	overlap	sections with the name *name@func* must be combined to one section *name*, according to the rules for *func* obtained from the call graph
S	separate	overlap	multiple sections can have the same name and they may relocated at unrelated addresses
N	now	when	section is located before normal sections (without N or P)
P	postpone	when	section is located after normal sections (without N or P)

Table I–1: Section types

5.3.3 SA COMMAND

SA_command	::=	”SA” section_number ”,” [MAU_boundary]? [”,” page_size]? ”.’
MAU_boundary	::=	expression
page_size	::=	expression

The MAU boundary value forces the relocator to align a section on the number of MAUs specified. If page_size is present the relocator checks that the section does not exceed a page boundary limit when it is relocated.

5.4 SYMBOLIC NAME DECLARATION AND TYPE DEFINITION

5.4.1 NI COMMAND

The NI command defines an internal symbol. An internal symbol is visible outside the module. Thus it may resolve an undefined external in another module.

NI_command ::= ”N” I_variable ”,” char_string ”.”

The NI_command must precede any reference to the I_variable in a module. There may not be more than one I_variable with the same name or number.

5.4.2 NX COMMAND

The NX command defines an external symbol which is undefined in the current module. The NX command must precede all occurrences of the corresponding X variable.

```
NX_command ::= ”N” X_variable ”,” char_string ”.”
```

The unresolved reference corresponding to an NX–command can be resolved by an internal symbol definition (NI_command) in another module.

5.4.3 NN COMMAND

The NN command defines a local name which may be used for defining a name of a local symbol in a module or a name in a type definition.

A name defined with an NN command is not visible outside the scope of the module. The NN command must precede all occurrences of the corresponding N variable.

```
NN_command ::= ”N” N_variable ”,” char_string ”.”
```

5.4.4 AT COMMAND

The attribute command may be used to define debugging related information of a symbol, such as the symbol type number. Level 2 of the standard does not prescribe the contents of the optional fields of the AT command. The language dependent layer (level 3) describes how these fields can be used to pass high–level symbol information with the AT command.

```
AT_command     ::=  ”AT” variable ”,” type_table_entry [ ”,” lex_level
                    [ ”,” hex_number ]* ]? ”.”

variable       ::=  I_variable | N_variable | X_variable
```

type_table_entry	::=	hex_number
lex_level	::=	hex_number

The type_table entry is a type number introduced with a type command (TY). References to type numbers in the AT command may precede the definition of the type in the TY command.

The meaning of the lex_level field is defined at layer 3 or higher. The same applies to the optional hex_number fields.

5.4.5 TY COMMAND

The TY–command defines a new type table entry. The type number introduced by the type command can be seen as a reference index to this type. The TY–command defines the relation between the newly introduced type and other types that are defined in other places in the object module. It also establishes a relation between a new type index and symbols (N_variable).

TY_command	::= ”TY” type_table_entry [”,” parameter]+ ”.”
type_table_entry	::= hex_number
parameter	::= hex_number | N_variable | ”T” type_table_entry

Level 2 does not define the semantics of the parameters. These are defined at level 3, the language layer. A linkage editor which does not have knowledge of the semantics of the parameter in a type command can still perform type comparison: Two types are considered to compare equal when the following conditions hold:

- both types have an equal number of parameters.
- the numeric values in the types are equal
- N_variables in both types have the same name
- the type entries referenced from both types compare equal

Variable N0 is supposed to compare equal to any other name.

Type table entry T0 is supposed to compare equal to any other type.

5.5 VALUE ASSIGNMENT

5.5.1 AS COMMAND

The assignment command assigns a value to a variable.

AS_command ::= ”AS” MUFOM_variable ”,” expression ”.”

5.6 LOADING COMMANDS

The contents of a section is either absolute data (code) or relocatable data (code). Absolute data can be loaded with the LD command. The address where loading takes place depends on the value of the P–variable belonging to the section. Data which is contiguous in a LD command is supposed to be loaded contiguously in memory.

If data is not absolute it contains expressions which must be evaluated by the expression evaluator. The LR command allows a relocation expression to be part of the loading command.

5.6.1 LD COMMAND

LD_command ::= ”LD” [hex_digit]+ ”.”

The constants loaded with the LD command are loaded with the most significant part first.

5.6.2 IR COMMAND

A relocation base is an expression which can be associated with a relocation letter. This relocation letter can be used in subsequent load relocate commands.

```
IR_command          ::=   ”IR” relocation_letter ”,” relocation_base
                          [ ”,” number_of_bits ]? ”.”

relocation_letter   ::=   nonhex_letter

relocation_base     ::=   expression
```

number_of_bits	::=	expression

Example:

```
IRV,X20,16.
ITM,R2,40,+,8.
```

The number_of_bits must be less than or equal to the number of bits per address, which is the product of the number of MAUs per address and the number of bits per MAU, both of which are specified in the AD command. If the number_of_bits is not specified it equals the number of bits per address.

5.6.3 LR COMMAND

LR_command	::=	”LR” [load_item]+ ”.”
load_item	::=	relocation_letter offset ”,” | load_constant |
		”(” expression [”,” number_of_MAUs]? ”)”
load_constant	::=	[hex_digit]+
number_of_MAUs	::=	expression

Examples:

```
LR002000400060.
LRT80,0020.
LR(R2,100,+,4).
```

The first example shows immediate constants which may be loaded as a part of an LR command.

The second example shows the use of the relocation base defined in the previous paragraph, followed by a constant.

The third example shows how the value of the expression R2 + 100 is used to load 4 MAUs.

The three commands in this example may be combined into one LR command:

```
LR002000400060T80,0020(R2,100,+,4).
```

5.6.4 RE COMMAND

The replicate command defines the number of times a LR command must be replicated:

RE_command ::= ”RE” expression ”.”

The LR command must immediately follow the RE command.

Example:

```
RE04.
LR(R2,200,+,4).
```

The commands above load 16 MAUs: 4 times the 4 MAU value of R2 + 200

5.7 LINKAGE COMMANDS

5.7.1 RI COMMAND

The retain internal symbol command indicates that the symbolic information of an NI command must be retained in the output file.

RI_command ::= ”R” I_variable [”,” level_number]? ”.”

level_number ::= hex_number

5.7.2 WX COMMAND

The weak external command flags a previously defined external (NX_command) as weak. This means that if the external remains unresolved, the value of the expression in the WX command is assigned to the X variable.

WX_command ::= ”W” X_variable [”,” default_value]? ”.”

default_value::= expression

5.7.3 LI COMMAND

The LI command specifies a default library search list. The library names specified in the LI_command are searched for unresolved references.

```
LI_command ::= ”LI” char_string [ ”,” char_string ]* ”.”
```

5.7.4 LX COMMAND

The LX command specifies a library to search for a named unresolved variable.

```
LX_command     ::= ”L” X_variable [ ”,” char_string ]+ ”.”
```

The paragraphs above showed the commands and operators as ASCII strings. In an object file they are binary encoded. The following tables show the binary representation.

6 MUFOM FUNCTIONS

The following table lists the first byte of MUFOM elements. Each value between 0 and 255 classifies the MUFOM language element that follows, or it is a language element itself. E.g. numbers outside the range 0–127 are preceded by a length field: 0x82 specifies that a 2 byte integer follows. 0xE4 is the function code for the LR command.

Overview of first byte of MUFOM language elements

Value	Description
0x00 – 0x7F	Start of regular string, or one byte numbers ranging from 0 – 127
0x80	Code for omitted optional number field
0x81 – 0x88	Numbers outside the range 0 – 127
0x89 – 0x8F	Unused
0x90 – 0xA0	User defined function codes
0xA0 – 0xBF	MUFOM function codes
0xC0	Unused
0xC1 – 0xDA	MUFOM letters
0xDB – 0xDF	Unused
0xE0 – 0xF9	MUFOM commands
0xFA – 0xFF	Unused

Table I–2: Overview of first byte of MUFOM language elements

Binary encoding of MUFOM letters and function codes

Function code		Identifiers	
Function	**code**	**Letter**	**code**
@F	0xA0		
@T	0xA1	A	0xC1
@ABS	0xA2	B	0xC2
@NEG	0xA3	C	0xC3
@NOT	0xA4	D	0xC4
+	0xA5	E	0xC5

Function code		Identifiers	
Function	**code**	**Letter**	**code**
–	0xA6	F	0xC6
/	0xA7	G	0xC7
*	0xA8	H	0xC8
@MAX	0xA9	I	0xC9
@MIN	0xAA	J	0xCA
@MOD	0xAB	K	0xCB
<	0xAC	L	0xCC
>	0xAD	M	0xCD
=	0xAE	N	0xCE
!= <>	0xAF	O	0xCF
@AND	0xB0	P	0xD0
@OR	0xB1	Q	0xD1
@XOR	0xB2	R	0xD2
@EXT	0xB3	S	0xD3
@INS	0xB4	T	OxD4
@ERR	0xB5	U	0xD5
@IF	0xB6	V	0xD6
@ELSE	0xB7	W	0xD7
@END	0xB8	X	0xD8
@ISDEF	0xB9	Y	0xD9
		Z	0xDA

Table I–3: Binary encoding of MUFOM letters and function codes

MUFOM Command codes

Command	Code	Description
MB	0xE0	Module begin
ME	0xE1	Module end
AS	0xE2	Assign
IR	0xE3	Inititialize relocation base

Command	Code	Description
LR	0xE4	Load with relocation
SB	0xE5	Section begin
ST	0xE6	Section type
SA	0xE7	Section alignment
NI	0xE8	Internal name
NX	0xE9	External name
CO	0xEA	Comment
DT	0xEB	Date and time
AD	0xEC	Address description
LD	0xED	Load
CS (with sum)	0xEE	Checksum followed by sum value
CS	0xEF	Checksum (reset sum to 0)
NN	0xF0	Name
AT	0xF1	Attribute
TY	0xF2	Type
RI	0xF3	Retain internal symbol
WX	0xF4	Weak external
LI	0xF5	Library search list
LX	0xF6	Library external
RE	0xF7	Replicate
SC	0xF8	Scope definition
LN	0xF9	Line number
	0xFA	Undefined
	0xFB	Undefined
	0xFC	Undefined
	0xFD	Undefined
	0xFE	Undefined
	0xFF	Undefined

Table I–4: MUFOM Command codes

APPENDIX J

MOTOROLA S–RECORDS

TASKING

APPENDIX J

With the **–f2** option the locator produces output in Motorola S–record format with three types of S–records: S0, S2 and S8. With the **–f2S1** or **–f2S3** option you can force other types of S–records. They have the following layout:

S0 – record

’S’ ’0’ *<length_byte>* <2 bytes 0> *<comment>* *<checksum_byte>*

A locator generated S–record file starts with a S0 record with the following contents:

length_byte	: $17
comment	: DSP563xx/6xx locator
checksum	: $0A

```
        D S P 5 6 3 x x / 6 x x   l o c a t o r
S017000044535035363378782F367878206C6F6361746F720A
```

The S0 record is a comment record and does not contain relevant information for program execution.

The length_byte represents the number of bytes in the record, not including the record type and length byte.

The checksum is calculated by first adding the binary representation of the bytes following the record type (starting with the length_byte) to just before the checksum. Then the one’s complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0FFH.

S1 – record

With the **–f2S1** option of the locator, the actual program code and data is supplied with S1 records, with the following layout:

’S’ ’1’ *<length_byte>* *<address>* *<code bytes>* *<checksum_byte>*

This record is used for 2–byte addresses.

Example:

```
S1130250F03EF04DF0ACE8A408A2A013EDFCDB00E6
  | |   |                             |_ checksum
  | |   |_ code
  | |_ address
  |_ length
```

The locator has an option that controls the length of the output buffer for generating S1 records. The default buffer length is 32 code bytes.

The checksum calculation of S1 records is identical to S0.

S2 – record

With the **–f2S2** option of the locator, which is the default, the actual program code and data is supplied with S2 records, with the following layout:

’S’ ’2’ *<length_byte> <address> <code bytes> <checksum_byte>*

For the DSP56xxx the locator generates 3–byte addresses.

Example:

```
S213FF002000232222754E00754F04AF4FAE4E22BF
  | |     |                             |_ checksum
  | |     |_ code
  | |_ address
  |_ length
```

The locator has an option that controls the length of the output buffer for generating S2 records. The default buffer length is 32 code bytes.

The checksum calculation of S2 records is identical to S0.

S3 – record

With the **–f2S3** option of the locator, the actual program code and data is supplied with S3 records, with the following layout:

’S’ ’3’ *<length_byte> <address> <code bytes> <checksum_byte>*

This record is used for 4–byte addresses.

Example:

```
S3070000FFFE6E6825
  | |       |   |_ checksum
  | |       |_ code
  | |_ address
  |_ length
```

The locator has an option that controls the length of the output buffer for generating S3 records.

The checksum calculation of S3 records is identical to S0.

S7 – record

With the **–f2S3** option of the locator, at the end of an S–record file, the locator generates an S7 record, which contains the program start address. S7 is the corresponding termination record for S3 records.

Layout:

’S’ ’7’ *<length_byte> <address> <checksum_byte>*

Example:

```
S70500006E6824
  | |       |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S7 records is identical to S0.

S8 – record

With the **–f2S2** option of the locator, which is the default, at the end of an S–record file, the locator generates an S8 record, which contains the program start address.

Layout:

’S’ ’8’ *<length_byte> <address> <checksum_byte>*

Example:

```
S804FF0003F9
  | |     |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S8 records is identical to S0.

S9 – record

With the **–f2S1** option of the locator, at the end of an S–record file, the locator generates an S9 record, which contains the program start address. S9 is the corresponding termination record for S1 records.

Layout:

’S’ ’9’ *<length_byte> <address> <checksum_byte>*

Example:

```
S9030210EA
  | |   |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S9 records is identical to S0.

APPENDIX K

INTEL HEX RECORDS

TASKING

APPENDIX K

Intel Hex records describe the hexadecimal object file format for 8–bit, 16–bit and 32–bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8–, 16, or 32–bit formats)
- End of File Record (8–, 16, or 32–bit formats)
- Extended Segment Address Record (16, or 32–bit formats)
- Start Segment Address Record (16, or 32–bit formats)
- Extended Linear Address Record (32–bit format only)
- Start Linear Address Record (32–bit format only)

For the DSP56xxx the locator generates records in the 32–bit format (4–byte addresses).

General Record Format

In the output file, the record format is:

:	*length*	*offset*	*type*	*content*	*checksum*

Where:

:	is the record header.
length	is the record length which specifies the number of bytes of the *content* field. This value occupies one byte (two hexadecimal digits). The locator outputs records of 255 bytes (32 hexadecimal digits) or less; that is, *length* is never greater than FFH.
offset	is the starting load offset specifying an absolute address in memory where the data is to be located when loaded by a tool. This field is two bytes long. This field is only used for Data Records. In other records this field is coded as four ASCII zero characters ('0000').
type	is the record type. This value occupies one byte (two hexadecimal digits). The record types are:

Byte Type	Record type
00	Data
01	End of File
02	Extended segment address (not used)
03	Start segment address (not used)
04	Extended linear address (32–bit)
05	Start linear address (32–bit)

content	is the information contained in the record. This depends on the record type.
checksum	is the record checksum. The locator computes the checksum by first adding the binary representation of the previous bytes (from *length* to *content*). The locator then computes the result of sum modulo 256 and subtracts the remainder from 256 (two's complement). Therefore, the sum of all bytes following the header is zero.

Extended Linear Address Record

The Extended Linear Address Record specifies the two most significant bytes (bits 16–31) of the absolute address of the first data byte in a subsequent Data Record:

:	02	0000	04	*upper_address*	*checksum*

The 32–bit absolute address of a byte in a Data Record is calculated as:

(*address* + *offset* + *index*) modulo 4G

where:

address	is the base address, where the two most significant bytes are the *upper_address* and the two least significant bytes are zero.
offset	is the 16–bit offset from the Data Record.
index	is the index of the data byte within the Data Record (0 for the first byte).

Example:

```
:0200000400FFFB
 | |   | |   |_ checksum
 | |   | |_ upper_address
 | |   |_ type
 | |_ offset
 |_ length
```

Data Record

The Data Record specifies the actual program code and data.

:	*length*	*offset*	00	*data*	*checksum*

The *length* byte specifies the number of *data* bytes. The locator has an option that controls the length of the output buffer for generating Data records. The default buffer length is 32 bytes.

The *offset* is the 16–bit starting load offset. Together with the address specified in the Extended Address Record it specifies an absolute address in memory where the data is to be located when loaded by a tool.

Example:

```
:0F00200000232222754E00754F04AF4FAE4E22C3
 | |   | |                             |_ checksum
 | |   | |_ data
 | |   |_ type
 | |_ offset
 |_ length
```

Start Linear Address Record

The Start Linear Address Record contains the 32–bit program execution start address.

Layout:

:	04	0000	05	*address*	*checksum*

Example:

```
:0400000500FF0003F5
 | |   | |       |_ checksum
 | |   | |_ address
 | |   |_ type
 | |_ offset
 |_ length
```

End of File Record

The hexadecimal file always ends with the following end–of–file record:

```
:00000001FF
 | |   | |_ checksum
 | |   |_ type
 | |_ offset
 |_ length
```

INDEX

INDEX

TASKING

INDEX