MA008-002-00-00
Doc. ver.: 1.13

8051 v7.2

C Cross-Compiler User's Manual

TASKING
Embedded software development from Altium

A publication of

Altium BV

Documentation Department

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

CONTENTS

TABLE OF CONTENTS

TASKING

CONTENTS

LANGUAGE IMPLEMENTATION 3-1

CONTENTS

CONTENTS

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the TASKING C 8051 Cross–Compiler. It assumes that you are familiar with the C language.

MANUAL STRUCTURE

Related Publications
Conventions Used In This Manual

Chapters

1. Software Installation
   Describes the installation of the C Cross–Compiler for the 8051.

2. Overview
   Provides an overview of the TASKING 8051 toolchain and gives you some familiarity with the different parts of it and their relationship. A sample session explains how to build an 8051 application from your C file.

3. Language Implementation
   Concentrates on the approach of the 8051 architecture and describes the language implementation. The C language itself is not described in this document. We recommend: “The C Programming Language” (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall).

4. Compiler Use
   Deals with compiler invocation, command line options and pragmas.

5. Compiler Diagnostics
   Describes the exit status and error/warning messages of the compilers.

6. Libraries
   Contains the library functions supported by the compilers and describes their interface and ’header’ files.

7. Run–time Environment
   Describes the run–time environment for a C–51 application. It deals with items like register usage, assembly language interfacing, C startup code, interrupt handlers, stack/heap size and floating point mathematic.

Appendices

A. MISRA C
   Supported and unsupported MISRA C rules.

B. SFR Definition File
   Contains an example of a Special Function Register definition file.

C. Restrictions for the 80751 and the 80752
   Contains the restrictions of the **cc51** for certain 8051 derivatives.

D. Converting PL/M–51 Applications to C–51
   Describes how to convert PL/M–51 applications to C–51.

E. CPU Functional Problems
   Describes how the 8051 toolchain can bypass some functional problems of the CPU.

F. Migration from Keil, Franklin or Archimedes
   Describes how you can migrate your C–51 application from the Keil, Franklin or Archimedes compiler to the TASKING C–51 compiler (**cc51**).

RELATED PUBLICATIONS

- The C Programming Language (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ANSI X3.159–1989 standard [ANSI]
- 8051 Cross–Assembler, Linker, Utilities User's Manual [TASKING, MA008–010–00–00]
- 8051 CrossView Pro Debugger User's Manual [TASKING, MA008–041–00–00]

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.

MANUAL STRUCTURE

CHAPTER 1

SOFTWARE INSTALLATION

TASKING

1 CHAPTER

1.1 INTRODUCTION

This chapter guides you through the procedures to install the software on a Windows system or on a Linux or UNIX host.

The software for Windows has two faces: a graphical interface (Embedded Development Environment) and a command line interface. The Linux and UNIX software have only a command line interface.

After the installation, it is explained how to configure the software and how to install the license information that is needed to actually use the software.

1.2 SOFTWARE INSTALLATION

1.2.1 INSTALLATION FOR WINDOWS

1. Start Windows 95/98/XP/NT/2000, if you have not already done so.
2. Insert the CD–ROM into the CD–ROM drive.

   If the TASKING Setup dialog box appears, proceed with Step 5.
3. Click the **Start** button and select **Run...**
4. In the dialog box type **`d:\setup`** (substitute the correct drive letter for your CD–ROM drive) and click on the **OK** button.

   The TASKING Setup dialog box appears.
5. Select a product and click on the **Install** button.
6. Follow the instructions that appear on your screen.

You can find your serial number on the invoice, delivery note, or picking slip delivered with the product.

7. License the software product as explained in section 1.4, *Licensing TASKING Products*.

1.2.2 INSTALLATION FOR LINUX

Each product on the CD–ROM is available as an RPM package, Debian package and as a gzipped tar file. For each product the following files are present:

```
SWproduct-version-RPMrelease.i386.rpm
swproduct_version-release_i386.deb
SWproduct-version.tar.gz
```

These three files contain exactly the same information, so you only have to install one of them. When your Linux distribution supports RPM packages, you can install the `.rpm` file. For a Debian based distribution, you can use the `.deb` file. Otherwise, you can install the product from the `.tar.gz` file.

RPM Installation

1. In most situations you have to be ”root” to install RPM packages, so either login as ”root”, or use the **su** command.
2. Insert the CD–ROM into the CD–ROM drive. Mount the CD–ROM on a directory, for example `/cdrom`. See the Linux manual pages about **mount** for details.
3. Go to the directory on which the CD–ROM is mounted:

   ```
   cd /cdrom
   ```

4. To install or upgrade all products at once, issue the following command:

   ```
   rpm -U SW*.rpm
   ```

   This will install or upgrade all products in the default installation directory `/usr/local`. Every RPM package will create a single directory in the installation directory.

   The RPM packages are 'relocatable', so it is possible to select a different installation directory with the **--prefix** option. For instance when you want to install the products in `/opt`, use the following command:

   ```
   rpm -U --prefix /opt SW*.rpm
   ```

For Red Hat 6.0 users: The **--prefix** option does not work with RPM version 3.0, included in the Red Hat 6.0 distribution. Please upgrade to RPM verion 3.0.3 or higher, or use the `.tar.gz` file installation described in the next section if you want to install in a non–standard directory.

Debian Installation

1. Login as a user.

   Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as ”root” or use the **su** command.

2. Insert the CD–ROM into the CD–ROM drive. Mount the CD–ROM on a directory, for example `/cdrom`. See the Linux manual pages about **mount** for details.

3. Go to the directory on which the CD–ROM is mounted:

   ```
   cd /cdrom
   ```

4. To install or upgrade all products at once, issue the following command:

   ```
   dpkg –i sw*.deb
   ```

   This will install or upgrade all products in a subdirectory of the default installation directory `/usr/local`.

Tar.gz Installation

1. Login as a user.

   Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as ”root” or use the **su** command.

2. Insert the CD–ROM into the CD–ROM drive. Mount the CD–ROM on a directory, for example `/cdrom`. See the Linux manual pages about **mount** for details.

3. Go to the directory on which the CD–ROM is mounted:

   ```
   cd /cdrom
   ```

4. To install the products from the `.tar.gz` files in the directory `/usr/local`, issue the following command for each product:

   tar xzf SW*product–version***.tar.gz –C /usr/local**

   Every `.tar.gz` file creates a single directory in the directory where it is extracted.

1.2.3 INSTALLATION FOR UNIX HOSTS

1. Login as a user.

   Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as ”root” or use the **su** command.

   If you are a first time user, decide where you want to install the product. By default it will be installed in `/usr/local`.

2. For CD–ROM install: insert the CD–ROM into the CD–ROM drive. Mount the CD–ROM on a directory, for example `/cdrom`. Be sure to use a ISO 9660 file system with Rock Ridge extensions enabled. See the UNIX manual pages about **mount** for details.

   Or:

   For tape install: insert the tape into the tape unit and create a directory where the contents of the tape can be copied to. Consider the created directory as a temporary workspace that can be deleted after installation has succeeded. For example:

   ```
   mkdir /tmp/instdir
   ```

3. For CD–ROM install: go to the directory on which the CD–ROM is mounted:

   ```
   cd /cdrom
   ```

   For tape install: copy the contents of the tape to the temporary workspace using the following commands:

   ```
   cd /tmp/instdir
   tar xvf /dev/tape
   ```

   where *tape* is the name of your tape device.

   If you have received a tape with more than one product, use the non–rewinding device for installing the products.

4. Run the installation script:

   ```
   sh install
   ```

   and follow the instructions appearing on your screen.

First a question appears about where to install the software. The default answer is **/usr/local**. On certain sites you may want to select another location.

On some hosts the installation script asks if you want to install SW000098, the Flexible License Manager (FLEXlm). If you do not already have FLEXlm on your system, you must install it; otherwise the product will not work on those hosts. See section 1.4, *Licensing TASKING Products*.

If the script detects that the software has been installed before, the following messages appear on the screen:

```
        *** WARNING ***
SWxxxxxx xxxx.xxxx already installed.
Do you want to REINSTALL? [y,n]
```

Answering **n** (no) to this question causes installation to abort and the following message being displayed:

```
=> Installation stopped on user request <=
```

Answering **y** (yes) to this question causes installation to continue. And the final message will be:

```
Installation of SWxxxxxx xxxx.xxxx completed.
```

5. For tape install: remove the temporary installation directory with the following commands:

   ```
   cd /tmp
   rm -rf instdir
   ```

6. If you purchased a protected TASKING product, license the software product as explained in section 1.4, *Licensing TASKING Products*.

1.3 SOFTWARE CONFIGURATION

Now you have installed the software, you can configure both the Embedded Development Environment and the command line environment for Windows, Linux and UNIX.

1.3.1 CONFIGURING THE EMBEDDED DEVELOPMENT ENVIRONMENT

After installation, the Embedded Development Environment is automatically configured with default search paths to find the executables, include files and libraries. In most cases you can use these settings. To change the default settings, follow the next steps:

1. Double–click on the EDE icon on your desktop to start the Embedded Development Environment (EDE).
2. From the **Project** menu, select **Directories...**

   The Directories dialog box appears.
3. Fill in the following fields:
   - In the **Executable Files Path** field, type the pathname of the directory where the executables are located. The default directory is `$(PRODDIR)\bin`.
   - In the **Include Files Path** field, add the pathnames of the directories where the compiler and assembler should look for include files. The default directory is `$(PRODDIR)\include`. Separate pathnames with a semicolon (;).

     The first path in the list is the first path where the compiler and assembler look for include files. To change the search order, simply change the order of pathnames.
   - In the **Library Files Path** field, add the pathnames of the directories where the linker should look for library files. The default directory is `$(PRODDIR)\lib`. Separate pathnames with a semicolon (;).

     The first path in the list is the first path where the linker looks for library files. To change the search order, simply change the order of pathnames.

Instead of typing the pathnames, you can click on the **Configure...** button.

A dialog box appears in which you can select and add directories, remove them again and change their order.

1.3.2 CONFIGURING THE COMMAND LINE ENVIRONMENT

To facilitate the invocation of the tools from the command line (either using a Windows command prompt or using Linux or UNIX), you can set *environment variables*.

You can set the following variables:

Environment Variable	Description
ASMDIR	With this variable you specify one or more additional directories in which the macro preprocessor **mpp51** looks for include files.
CC51INC	With this variable you specify one or more additional directories in which the C compiler **cc51** looks for include files. The compiler first looks in these directories, then always looks in the default `include` directory relative to the installation directory.
CC51LIB	With this variable you specify one or more additional directories in which the linker **link51** looks for library files.
LM_LICENSE_FILE	With this variable you specify the location of the license data file. You only need to specify this variable if the license file is not on its default location (`c:\flexlm` for Windows, `/usr/local/flexlm/licenses` for UNIX).
PATH	With this variable you specify the directory in which the executables reside. This allows you to call the executables when you are not in the `bin` directory. Usually your system already uses the PATH variable for other purposes. To keep these settings, you need to add (rather than replace) the path. Use a semicolon (;) to separate pathnames.

Environment Variable	Description
TASKING_LIC_WAIT	If you set this variable, the tool will wait for a license to become available, if all licenses are taken. If you have not set this variable, the tool aborts with an error message. (Only useful with floating licenses)
TMPDIR	With this variable you specify the location where programs can create temporary files. Usually your system already uses this variable. In this case you do not need to change it.

Table 1–1: Environment variables

The following examples show how to set an environment variable using the CC51INC variable as an example.

See also section 4.3, *Include Files* in chapter *Compiler Use*.

Example Windows 95/98

Add the following line to your `autoexec.bat` file.

```
set CC51INC=c:\cc51\include
```

You can also type this line in a Command Prompt window but you will loose this setting after you close the window.

Example Windows NT

1. Right–click on the `My Computer` icon on your desktop and select **Properties**.

   The System Properties dialog appears.

2. Select the **Environment** tab.

3. In the **Variable** edit field enter:

   ```
   CC51INC
   ```

4. In the **Value** edit field enter:

   ```
   c:\cc51\include
   ```

5. Click on the **Set** button, then click **OK**.

Example Windows XP / 2000

1. Right–click on the **My Computer** icon on your desktop and select **Properties**.

   The System Properties dialog appears.

2. Select the **Advanced** tab and click on the **Environment Variables** button.

   The Environment Variables dialog appears.

3. In the **System variables** field, click on the **New** button.

   The New System Variable dialog appears.

4. In the **Variable name** field enter:

   ```
   CC51INC
   ```

5. In the **Variable value** field enter:

   ```
   c:\cc51\include
   ```

6. Click on the **OK** button to accept the changes and close the dialogs.

Example for UNIX

Enter the following line (C–shell):

```
setenv CC1INC /usr/local/cc51/include
```

1.4 LICENSING TASKING PRODUCTS

TASKING products are protected with license management software (FLEXlm). To use a TASKING product, you must install the license key provided by TASKING for the type of license purchased.

You can run TASKING products with a node–locked license or with a floating license. When you order a TASKING product determine which type of license you need (UNIX products only have a floating license).

Node–locked license (PC only)

This license type locks the software to one specific PC so you can use the product on that particular PC only.

Floating license

This license type manages the use of TASKING product licenses among users at one site. This license type does not lock the software to one specific PC or workstation but it requires a network. The software can then be used on any computer in the network. The license specifies the number of users who can use the software simultaneously. A system allocating floating licenses is called a **license server**. A license manager running on the license server keeps track of the number of users.

1.4.1 OBTAINING LICENSE INFORMATION

Before you can install a software license you must have a ”License Key” containing the license information for your software product. If you have not received such a license key follow the steps below to obtain one. Otherwise, you can install the license.

Windows

1. Run the License Administrator during installation and follow the steps to **Request a license key from Altium by E–mail**.
2. E–mail the license request to your local TASKING sales representative. The license key will be sent to you by E–mail.

UNIX

1. If you need a floating license on UNIX, you must determine the host ID and host name of the computer where you want to use the license manager. Also decide how many users will be using the product. See section 1.4.5, *How to Determine the Host ID* and section 1.4.6, *How to Determine the Host Name*.

2. When you order a TASKING product, provide the host ID, host name and number of users to your local TASKING sales representative. The license key will be sent to you by E–mail.

1.4.2 INSTALLING NODE-LOCKED LICENSES

If you do not have received your license key, read section 1.4.1, *Obtaining License Information*, before continuing.

1. Install the TASKING software product following the installation procedure described in section 1.2.1, *Installation for Windows*, if you have not done this already.

2. Create a license file by importing a license key or create one manually:

Import a license key

During installation you will be asked to run the License Administrator. Otherwise, start the License Administrator (**licadmin.exe**) manually.

In the License Administrator follow the steps to **Import a license key received from Altium by E–mail**. The License Administrator creates a license file for you.

Create a license file manually

If you prefer to create a license file manually, create a file called "`license.dat`" in the `c:\flexlm` directory, using an ASCII editor and insert the license key information received by E–mail in this file. This file is called the "license file". If the directory `c:\flexlm` does not exist, create the directory.

If you wish to install the license file in a different directory, see section 1.4.4, *Modifying the License File Location*.

If you already have a license file, add the license key information to the existing license file. If the license file already contains any SERVER lines, you must use another license file. See section 1.4.4, *Modifying the License File Location*, for additional information.

The software product and license file are now properly installed.

1.4.3 INSTALLING FLOATING LICENSES

If you do not have received your license key, read section 1.4.1, *Obtaining License Information*, before continuing.

1. Install the TASKING software product following the installation procedure described earlier in this chapter on each computer or workstation where you will use the software product.
2. On each PC or workstation where you will use the TASKING software product the location of a license file must be known, containing the information of all licenses. Either create a local license file or point to a license file on a server:

Add a licence key to a local license file

A local license file can reduce network traffic.

On Windows, you can follow the same steps to import a license key or create a license file manually, as explained in the previous section with the installation of a node–locked license.

On UNIX, you have to insert the license key manually in the license file. The default location of the license file `license.dat` is in directory `/usr/local/flexlm/licenses` for UNIX.

If you wish to install the license file in a different directory, see section 1.4.4, *Modifying the License File Location*.

If you already have a license file, add the license key information to the existing license file. If the license file already contains any SERVER lines, make sure that the number of SERVER lines and their contents match, otherwise you must use another license file. See section 1.4.4, *Modifying the License File Location*, for additional information.

Point to a license file on the server

Set the environment variable **LM_LICENSE_FILE** to "*port@host*", where *host* and *port* come from the SERVER line in the license file. On Windows, you can use the License Administrator to do this for you. In the License Administrator follow the steps to **Point to a FLEXlm License Server to get your licenses**.

3. If you already have installed FLEXlm v8.4 or higher (for example as part of another product) you can skip this step and continue with step 4. Otherwise, install SW000098, the Flexible License Manager (FLEXlm), on the license server where you want to use the license manager.

It is not recommended to run a license manager on a Windows 95 or Windows 98 machine. Use Windows XP, NT or 2000 instead, or use UNIX or Linux.

4. If FLEXlm has already been installed as part of a non–TASKING product you have to make sure that the `bin` directory of the FLEXlm product contains a copy of the **Tasking** daemon. This file is present on every product CD that includes FLEXlm, in directory `licensing`.

5. On the license server also add the license key to the license file. Follow the same instructions as with "Add a license key to a local license file" in step 2.

See the FLEXlm PDF manual delivered with SW000098, which is present on each TASKING product CD, for more information.

1.4.4 MODIFYING THE LICENSE FILE LOCATION

The default location for the license file on Windows is:

```
c:\flexlm\license.dat
```

On UNIX this is:

```
/usr/local/flexlm/licenses/license.dat
```

If you want to use another name or directory for the license file, each user must define the environment variable **LM_LICENSE_FILE**.

If you have more than one product using the FLEXlm license manager you can specify multiple license files to the **LM_LICENSE_FILE** environment variable by separating each pathname (*lfpath*) with a '**;**' (on UNIX '**:**'):

Example Windows:

```
set LM_LICENSE_FILE=c:\flexlm\license.dat;c:\license.txt
```

Example UNIX:

```
setenv LM_LICENSE_FILE
/usr/local/flexlm/licenses/license.dat:/myprod/license.txt
```

If the license file is not available on these hosts, you must set **LM_LICENSE_FILE** to *port@host*; where *host* is the host name of the system which runs the FLEXlm license manager and *port* is the TCP/IP port number on which the license manager listens.

To obtain the port number, look in the license file at *host* for a line starting with "SERVER". The fourth field on this line specifies the TCP/IP port number on which the license server listens. For example:

```
setenv LM_LICENSE_FILE 7594@elliot
```

See the FLEXlm PDF manual delivered with SW000098, which is present on each TASKING product CD, for detailed information.

1.4.5 HOW TO DETERMINE THE HOST ID

The host ID depends on the platform of the machine. Please use one of the methods listed below to determine the host ID.

Platform	Tool to retrieve host ID	Example host ID
HP–UX	**lanscan** (use the station address without the leading '0x')	0000F0050185
Linux	**hostid**	11ac5702
SunOS/Solaris	**hostid**	170a3472
Windows	**licadmin** (License Administrator, or use **lmhostid**)	0060084dfbe9

Table 1–2: Determine the host ID

On Windows, the License Administrator (**licadmin**) helps you in the process of obtaining your license key.

If you do not have the program **licadmin** you can download it from our Web site at: http://www.tasking.com/support/flexlm/licadmin.zip . It is also on every product CD that includes FLEXlm, in directory `licensing`.

1.4.6 HOW TO DETERMINE THE HOST NAME

To retrieve the host name of a machine, use one of the following methods.

Platform	Method
UNIX	**hostname**
Windows NT	**licadmin** or: Go to the Control Panel, open "Network". In the "Identification" tab look for "Computer Name".
Windows XP/2000	**licadmin** or: Go to the Control Panel, open "System". In the "Computer Name" tab look for "Full computer name".

Table 1–3: Determine the host name

INSTALLATION

CHAPTER 2

OVERVIEW

2 CHAPTER

2.1 INTRODUCTION TO 8051 C CROSS-COMPILER

This manual provides a functional description of the TASKING 8051 C Cross–Compiler. This manual uses **cc51** (the name of the binary) as a shorthand notation for "TASKING 8051 C Compiler".

TASKING offers a complete toolchain for the 8051 family of microcontroller units and their derivatives. '8051' is used as a shorthand notation for this microcontroller. The toolchain contains a C compiler, a macro preprocessor, an assembler, a linker/locator, a library manager and a debugger.

Unlike other C–8051 compilers, **cc51** is not a general C compiler, adapted for the 8051, but it is dedicated to the microcontroller architecture of the 8051. This means that you can access all special features of the 8051 in C: multiple address spaces (with full pointer support), bit memory, special function registers (I/O ports), interrupt functions using bank switching and a number of built–in (inline) functions to access special 8051 instructions. And yet no compromise is made to the ANSI standard. It is a fast, single pass, optimizing compiler that generates extremely fast and compact code.

cc51 generates assembly source code using the Intel assembly language specification, which can be assembled with the TASKING 8051 Cross–Assembler (In this document we use **asm51** as a shorthand notation for 'TASKING 8051 Cross–Assembler').

You can link the generated object with other objects and libraries by using the TASKING **link51** linker/locator (In this document we use **link51** as a shorthand notation for 'TASKING **link51** linker/locator'). The software written in C can be debugged using a TASKING high–level language debugger. A list of supported platforms and emulators is available from TASKING.

Target Processors:

All 8051 derivatives. Special function registers can be accessed by means of user–definable 'sfrfile'.

2.2 GENERAL IMPLEMENTATION

This section describes the different phases of the compiler and the target independent optimizations.

2.2.1 COMPILER PHASES

During the compilation of a C program, a number of phases can be identified. These phases are divided into two groups, referred to as *frontend* and *backend*.

frontend:

The preprocessor phase:

File inclusion and macro substitution are done by the preprocessor before parsing of the C program starts. The syntax of the macro preprocessor is independent of the C syntax, but also described in the ANSI X3.159–1989 standard.

The scanner phase:

The scanner converts the preprocessor output to a stream of tokens.

The parser phase:

The tokens are fed to a parser for the C grammar. The parser performs a syntactic and semantic analysis of the program, and generates an intermediate representation of the program.

The frontend optimization phase:

Target processor independent optimization is performed by transforming the intermediate code. The next section discusses the frontend optimizations.

backend:

The backend optimization phase:

Performs target processor specific optimizations. Very often this means another transformation of the intermediate code and actions like register allocation techniques for variables, expression evaluation and the best usage of the addressing modes. The chapter *Language Implementation* discusses this item in more detail.

The code generator phase:

This phase converts the intermediate code to an internal instruction code, representing the 8051 assembly instructions.

The peephole optimizer phase:

This phase uses pattern matching techniques to perform peephole optimizations on the internal code (e.g. deleting obsolete moves). Another task of the peephole optimizer is to convert LJMP instructions to SJMP instructions (or to reverse the condition of conditional bit jump instructions), if the destination label is not within the REL range (–128 to 127 words). Finally, the peephole optimizer translates the internal instruction code into assembly code for **asm51**. The generated assembly does not contain any macros.

All phases (of both frontend and backend) are combined into one program: **cc51**. The compiler does not use any intermediate file for communication between the different phases of compilation. The backend part is not called for each C statement, but is started after a complete C function has been processed by the frontend (in memory), thus allowing more optimization. The compiler only requires one pass over the input file, resulting in relatively fast compilation.

2.2.2 FRONTEND OPTIMIZATIONS

The compiler performs the following optimizations on the intermediate code. They are independent of the target processor and the code generation strategy:

Constant folding

Expressions only involving constants are replaced by their result.

Expression rearrangement

Expressions are rearranged to allow more constant folding. E.g. `1+ (x-3)` is transformed into `x + (1-3)`, which can be folded.

Expression simplification

Multiplication by 0 or 1 and additions or subtractions of 0 are removed. Such useless expressions may be introduced by macros in C (#define), or by the compiler itself.

Logical expression optimization

Expressions involving '&&', '||' and '!' are interpreted and translated into a series of conditional jumps.

Loop rotation

With `for` and `while` loops, the expression is evaluated once at the 'top' and then at the 'bottom' of the loop. This optimization does not save code, but speeds up execution.

Switch optimization

A number of optimizations of a switch statement are performed, such as the deletion of redundant case labels or even the deletion of the switch.

Control flow optimization

By reversing jump conditions and moving code, the number of jump instructions is minimized. This reduces both the code size and the execution time.

Jump chaining

A conditional or unconditional jump to a label which is immediately followed by an unconditional jump may be replaced by a jump to the destination label of the second jump. This optimization does not save code, but speeds up execution.

Remove useless jumps

An unconditional jump to a label directly following the jump is removed. A conditional jump to such a label is replaced by an evaluation of the jump condition. The evaluation is necessary because it may have side effects.

Conditional jump reversal

A conditional jump over an unconditional jump is transformed into one conditional jump with the jump condition reversed. This reduces both the code size and the execution time.

Constant/copy propagation

A reference to a variable with known contents is replaced by those contents.

Common subexpression elimination

The compiler has the ability to detect repeated uses of the same (sub–) expression. Such a ”common” expression may be temporarily saved to avoid recomputation. This method is called *common subexpression elimination*, abbreviated CSE.

Dead code elimination

Unreachable code can be removed from the intermediate code without affecting the program. However, the compiler generates a warning message, because the unreachable code may be the result of a coding error.

Loop optimization

Invariant expressions may be moved out of a loop and expressions involving an index variable may be reduced in strength.

Loop unrolling

Eliminate short loops by replacing them with a number of copies.

Sharing of string literals and floating point constants

String literals and floating point constants are put in ROM memory. The compiler overlays identical strings (within the same module) and let them share the same space, thus saving ROM space. Likewise identical floating point constants are overlaid and allocated only once.

2.3 PROGRAM DEVELOPMENT FLOW

If you want to build a 8051 application you need to invoke the following programs directly, or via the control program:

- The C compiler (**cc51**), which generates an assembly source file from the file with suffix `.c` (or `.i`). The suffix of the compiler output file is `.src`, which is the default for **asm51**. However, you can direct the output to `stdout` with the **–n** option, or to another file with the **–o** option. C source lines can be intermixed with the generated assembly statements with the **–s** option. High level language debugging information can be generated with the **–g** option. You are advised not to use the **–g** option when inspecting the generated assembly source code, because it contains a lot of 'unreadable' high level language debug directives. **cc51** makes only one pass on every file. This pass checks the syntax, generates the code and performs code optimization.
- The corresponding cross–assembler (**asm51**), which processes the generated assembly source file into a relocatable object file with suffix `.obj`. A full assembly listing with suffix `.lst` is available after this stage.
- The **link51** linker/locator, which links the generated relocatable object files, startup code and C libraries. The result is a loadable file in `a.out` type (COFF) format. A full memory map is available at this stage.
- The **ieee51** program which formats an `a.out` type file into a debugger load file in IEEE–695 format.

The next figure explains the relationship between the different parts of the TASKING 8051 toolchain:

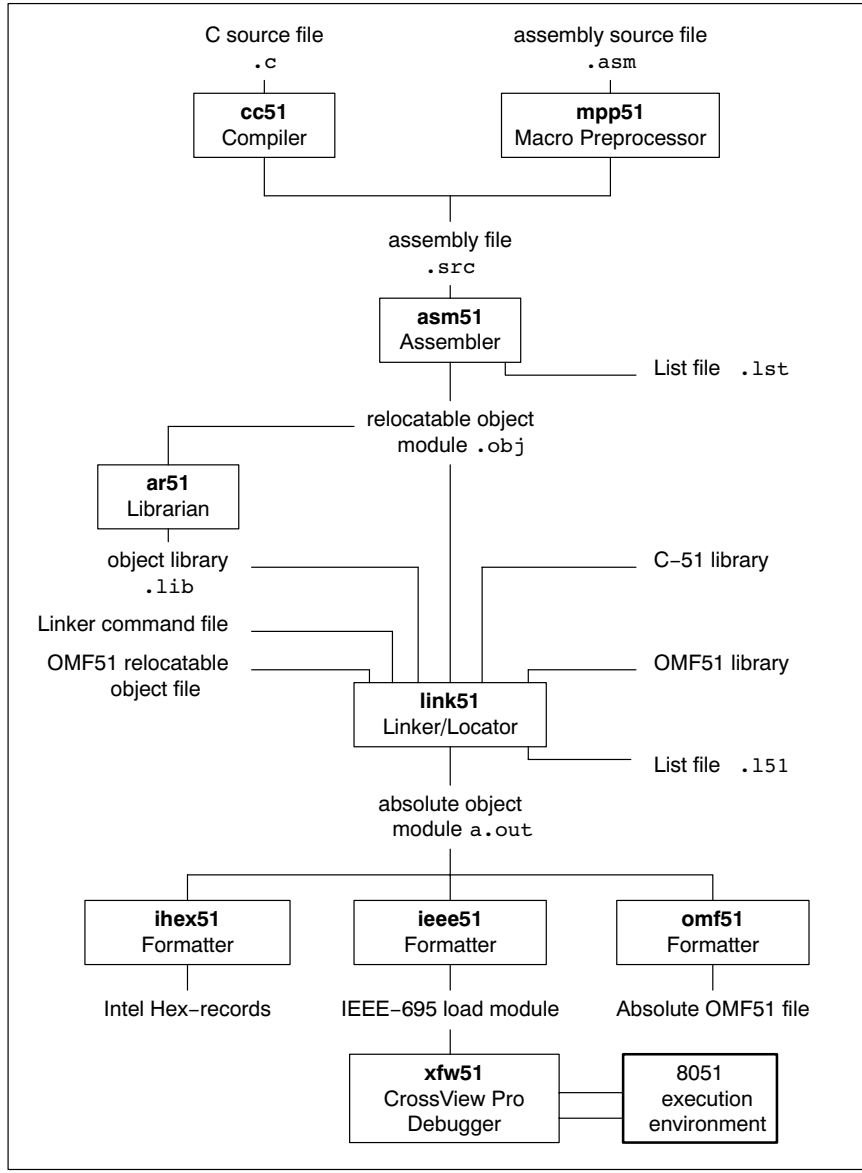


Figure 2–1: 8051 development flow

The **ihex51** program formats the `a.out` file into an Intel Hex format file. You can load this output file into an EPROM programmer.

The **omf51** program formats the `a.out` file into a (full symbol) absolute Intel OMF51 format file. This output file can be loaded into a debugger. See the appendices for examples.

The **ar51** program is a librarian facility. You can use this program to create and maintain object libraries.

The programs **ieee51**, **ihex51**, **omf51** and **ar51** are delivered with the **asm51**/**link51** package.

For a full description of all available formatter programs and other utilities, we refer to the *8051 Cross–Assembler User's Manual*.

The name of the 8051 CrossView Pro Debugger is **xfw51**. For more information check the *8051 CrossView Pro Debugger User's Manual*.

File extensions

The following table lists the file types used by the 8051 toolchain.

Extension	Description
Source files	
.c	C source file, input for the C compiler
.asm	Assembler source file, hand coded, input for the macro preprocessor
.sfr	Special function register file
Generated source files	
.src	Assembler source file, generated by the C compiler or macro preprocessor
Object files	
.obj	IEEE–695 relocatable object file, generated by the assembler
.lib	Object library file
.omf	OMF51 object file
.out	Relocatable linker output file
.abs	IEEE–695 absolute object file
.hex	Intel Hex absolute object file
.sre	Motorola S–record absolute object file
List files	
.lst	Assembler list file
.l51	Linker map file
Error list files	
.err	Compiler error messages file

Table 2–1: File extensions

2.4 WORKING WITH PROJECTS IN EDE

EDE is a complete project environment in which you can create and maintain project spaces and projects. EDE gives you direct access to the tools and features you need to create an application from your project.

A *project space* holds a set of projects and must always contain at least one project. Before you can create a project you have to setup a project space. All information of a project space is saved in a *project space file* (`.psp`):

- a list of projects in the project space
- history information

Within a project space you can create *projects*. Projects are bound to a target! You can create, add or edit files in the project which together form your application. All information of a project is saved in a *project file* (`.pjt`):

- the target for which the project is created
- a list of the source files in the project
- the options for the compiler, assembler, linker and debugger
- the default directories for the include files, libraries and executables
- the build options
- history information

When you build your project, EDE handles file dependencies and the exact sequence of operations required to build your application. When you push the **Build** button, EDE generates a makefile, including all dependencies, and builds your application.

Overview of steps to create and build an application

1. Create a project space
2. Add one or more projects to the project space
3. Add files to the project
4. Edit the files
5. Set development tool options
6. Build the application

2.5 START EDE

Start EDE

- Double–click on the EDE shortcut on your desktop.

  – or –

  Launch EDE via the program folder created by the installation program. Select **Start –> Programs –> TASKING *toolchain* –> EDE**.

Figure 2–2: EDE icon

The EDE screen contains a menu bar, a toolbar with command buttons, one or more windows (default, a window to edit source files, a project window and an output window) and a status bar.

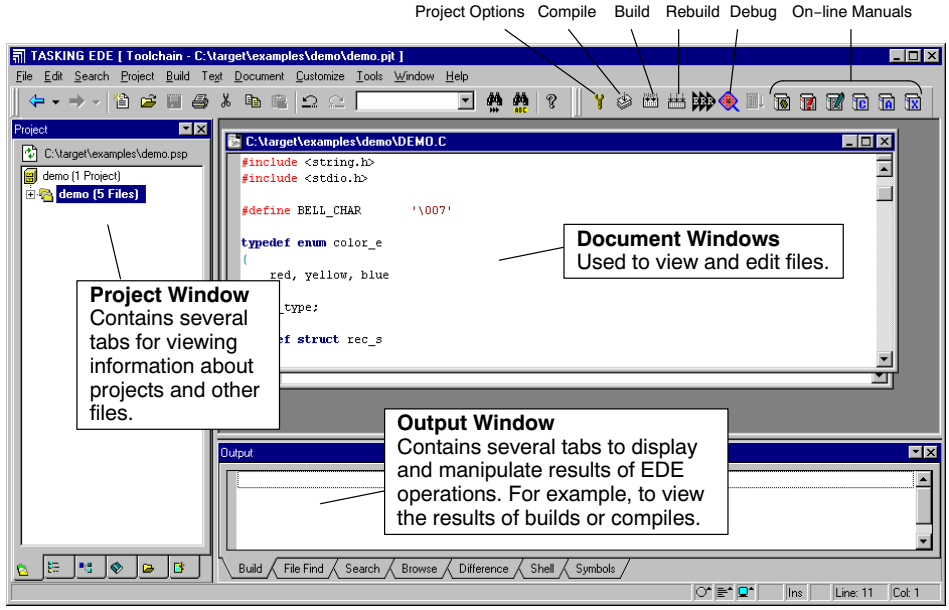


Figure 2–3: EDE desktop

2.6 USING THE SAMPLE PROJECTS

When you start EDE for the first time (see section 2.5, *Start EDE*), EDE opens with a ready defined project space that contains several sample projects. Each project has its own subdirectory in the `examples` directory. Each directory contains a file `readme.txt` with information about the example. The default project is called `demo.pjt` and contains a CrossView Pro debugger example.

Select a sample project

To select a project from the list of projects in a project space:

1. In the Project Window, right–click on the project you want to open.

   A menu appears.

2. Select **Set as Current Project**.

   The selected project opens.

3. Read the file `readme.txt` for more information about the selected sample project.

Building a sample project

To build the currently active sample project:

- Click on the **Execute 'Make' command** button.

*Once the files have been processed you can inspect the generated messages in the **Build** tab of the **Output** window.*

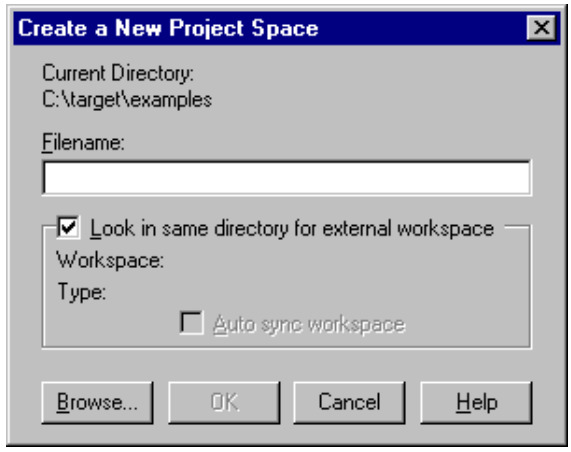
2.7 CREATE A NEW PROJECT SPACE WITH A PROJECT

Creating a project space is in fact nothing more than creating a project space file (`.psp`) in an existing or new directory.

Create a new project space

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

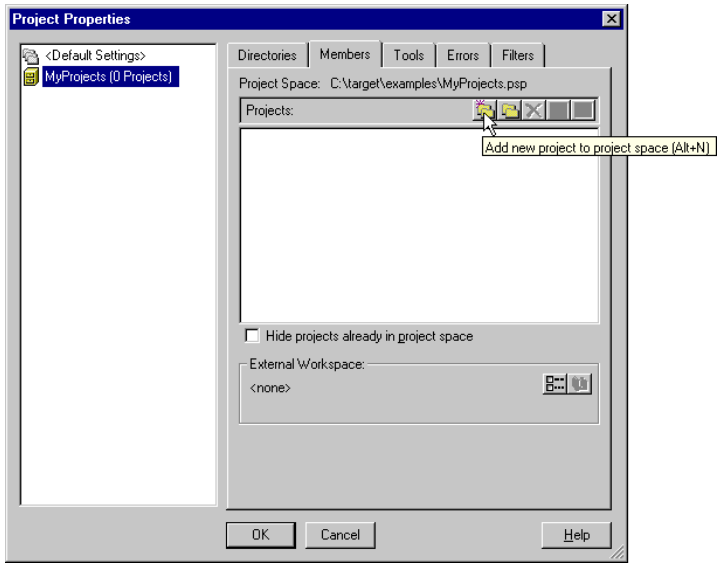
   The Create a New Project Space dialog appears.



2. In the the **Filename** field, enter a name for your project space (for example `MyProjects`). Click the **Browse** button to select a directory first and enter a filename.

3. Check the directory and filename and click **OK** to create the `.psp` file in the directory shown in the dialog.

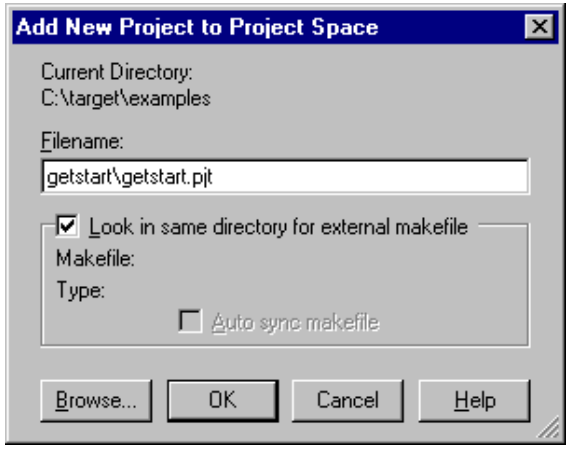
   A project space information file with the name `MyProjects.psp` *is created and the Project Properties dialog box appears with the project space selected.*



Add a new project to the project space

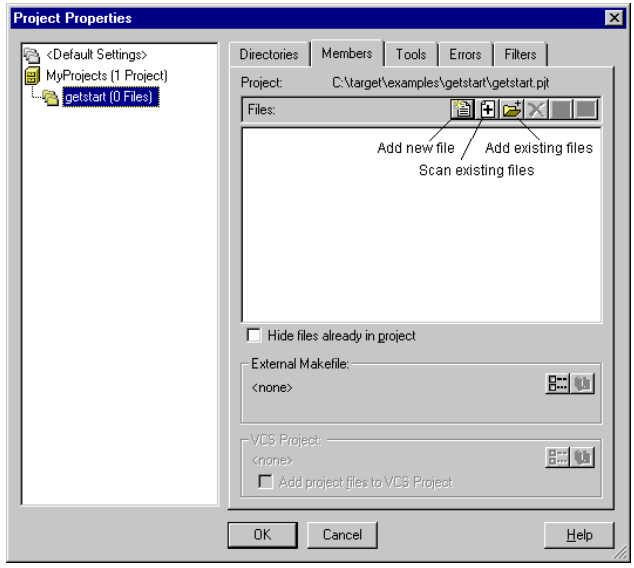
4. In the Project Properties dialog, click on the **Add new project to project space** button (see previous figure).

   The Add New Project to Project Space dialog appears.



5. Give your project a name, for example `getstart\getstart.pjt` (a directory name to hold your project files is optional) and click **OK**.

   A project file with the name `getstart.pjt` *is created in the directory* `getstart`*, which is also created. The Project Properties dialog box appears with the project selected.*

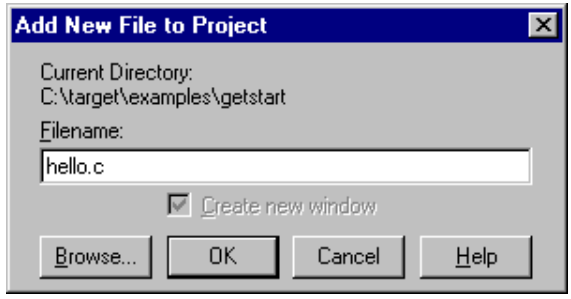


Add new files to the project

Now you can add all the files you want to be part of your project.

6. Click on the **Add new file to project** button.

   The Add New File to Project dialog appears.



7. Enter a new filename (for example `hello.c`) and click **OK**.

   A new empty file is created and added to the project. Repeat steps 6 and 7 if you want to add more files.

8. Click **OK**.

   The new project is now open. EDE loads the new file(s) in the editor in separate document windows.

   EDE automatically creates a *makefile* for the project (in this case `getstart.mak`). This file contains the rules to build your application. EDE updates the makefile every time you modify your project.

Edit your files

9. As an example, type the following C source in the `hello.c` document window:

```
#include <stdio.h>

void main(void)
{
    printf("Hello World!\n");
}
```

10. Click on the **Save the changed file <Ctrl–S>** button.

EDE saves the file.

2.8 SET OPTIONS FOR THE TOOLS IN THE TOOLCHAIN

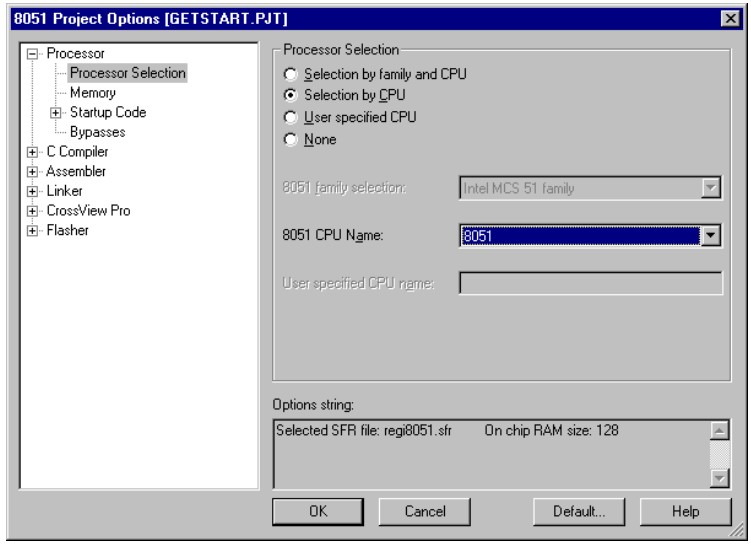
The next step in the process of building your application is to select a target processor and specify the options for the different parts of the toolchain, such as the C compiler, assembler, linker and debugger.

Select a target processor

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

   The Project Options dialog appears.

2. Expand the **Processor** entry and select **Processor Selection**.



3. Optionally select a **8051 family** to narrow the list of processors.

4. In the **8051 CPU name** list select your target processor (for example, **8051**).

5. Click **OK** to accept the new project settings.

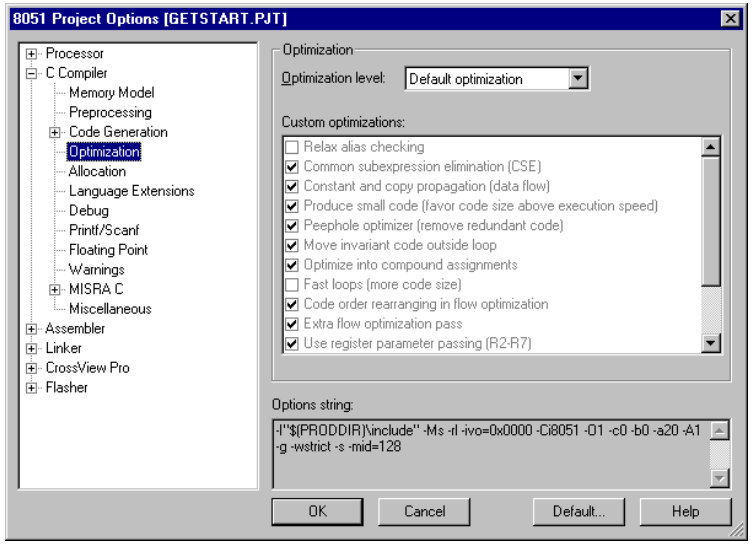
Set tool options

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

   *The Project Options dialog appears. Here you can specify options that are valid for the entire project. To overrule the project options for the currently active file instead, from the **Project** menu select **Current File Options...***

2. Expand the **C Compiler** entry.

   The C Compiler entry contains several pages where you can specify C compiler settings.



3. For each page make your changes. If you have made all changes click **OK**.

The **Cancel** button closes the dialog without saving your changes. With the **Default...** button you can restore the default project options (for the current page, or all pages in the dialog).

4. Make your changes for all other entries (Assembler, Linker, CrossView Pro, Flasher) of the Project Options dialog in a similar way as described above for the C compiler.

If available, the **Options string** field shows the command line options that correspond to your graphical selections.

2.9 BUILD YOUR APPLICATION

If you have set all options, you can actually compile the file(s). This results in an absolute IEEE–695 object file which is ready to be debugged.

Build your Application

To build the currently active project:

- Click on the **Execute 'Make' command** button.

The file is compiled, assembled, linked and located. The resulting file is `getstart.abs`.

The build process only builds files that are out–of–date. So, if you click **Make** again in this example nothing is done, because all files are up–to–date.

Viewing the Results of a Build

Once the files have been processed, you can see which commands have been executed (and inspect generated messages) by the build process in the **Build** tab of the **Output** window.

This window is normally open, but if it is closed you can open it by selecting the **Output** menu item in the **Window** menu.

Compiling a Single File

1. Select the window (document) containing the file you want to compile or assemble.
2. Click on the **Execute 'Compile' command** button. The following button is the execute Compile button which is located in the toolbar.

If you selected the file `hello.c`*, this results in the compiled and assembled file* `hello.obj`.

Rebuild your Entire Application

If you want to compile, assemble and link/locate all files of your project from scratch (regardless of their date/time stamp), you can perform a rebuild.

- Click on the **Execute 'Rebuild' command** button. The following button is the execute Rebuild button which is located in the toolbar.

2.10 HOW TO BUILD YOUR APPLICATION ON THE COMMAND LINE

If you are not using EDE, you can build your entire application on the command line. The easiest way is to use the *control program* **c51**.

1. In a text editor, write the file `hello.c` with the following contents:

```
#include <stdio.h>

void main(void)
{
    printf("Hello World!\n");
}
```

2. Build the file `getstart.abs`:

```
c51 -g -o getstart.abs hello.c
```

The control program calls all tools in the toolchain. The ***-v*** *option shows all the individual steps. The resulting file is* `getstart.abs`.

2.10.1 USING A MAKEFILE

The `examples` directory contains several subdirectories with example programs. Each subdirectory contains a `makefile` which can be processed by **mk51** to build the example. Also each subdirectory contains a `readme.txt` file with a description of how to build the example.

To build the `mesage` 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 `message` of the `examples` directory the current working directory.

   This directory contains a makefile for building the demo example. It uses the default **mk51** 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 **mk51**:

   ```
   mk51
   ```

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

   To see which commands are invoked by **mk51** without actually executing them, type:

   ```
   mk51 -n
   ```

   To remove all generated files type:

   ```
   mk51 clean
   ```

2.11 DEBUGGING YOUR APPLICATION

Once the files have been compiled with symbolic debug information enabled (option **–g**), assembled, linked, located and formatted they are ready for debugging.

Start CrossView Pro

- Click on the **Debug application** button.

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

How to Set the Communication Parameters of CrossView Pro ROM

When you use CrossView Pro ROM for the first time, you must setup the communication parameters.

To set the communication parameters:

1. From the **File** menu, select **Communication Setup...**

   The Communication Setup dialog appears.

2. In this dialog you need to identify the COM port (probably COM1: or COM2:) and the baud rate (9600 for RISM).

3. Click **OK** to close the dialog.

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**.

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 `message.abs` is now stepping through the high level language statements. Using the Accelerator bar 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 Manual* for more information.

CHAPTER 3

LANGUAGE IMPLEMENTATION

TASKING

CHAPTER 3

3.1 INTRODUCTION

The TASKING 8051 C cross–compiler offers a new approach to high–level language programming for the 8051 family. It conforms to the ANSI standard, but allows the user to control the I/O registers, bit memory, interrupts (register bank switch) and multiple address spaces of the 8051 in C.

This chapter describes the language implementation in relation to the 8051 architecture.

The extensions to the C language in **cc51** are:

_bit

You can use data type `bit` or `_bit` for the type definition of scalars and for the return type of functions.

_bitbyte

You can declare byte variables in the bit–addressable area as `_bitbyte`. You can access additional bits using the built–in functions `_getbit()` and `_putbit()`.

_sfrbit

Data type for the declaration of specific, absolute bits in special function registers or special absolute bits in the SFR address space.

_sfrbyte* / *_sfrword

Data types for the declaration of Special Function Registers.

_at

You can specify a variable to be at an absolute address.

_atbit

You can specify a variable to be at a bit offset within a bitaddressable variable.

_plmprocedure

Declaration of external PL/M–51 procedures.

_inline

Used for defining inline functions.

_noregaddr

You can specify a function to be independent of register banks.

storage types

Apart from a memory category (extern, static, ...) you can specify a storage type in each declaration. This way you obtain a memory model–independent addressing of variables in several address ranges of the 8051 (`_data`, `_bdat`, `_idat`, `_pdat`, `_xdat`, `_rom`).

memory-specific pointers

cc51 allows you to define pointers which point to a specific target memory. These types of pointers are very efficient and require only 1 or 2 bytes memory space.

mixed memory models

cc51 allows you to combine memory models by using a default memory model and assigning specific memory models to functions. For example, a program created in the large memory model can be accelerated, in which functions are partially distributed to the small model. The keywords you can use to specify a model for a function are: `_small`, `_aux`, `_large` and `_reentrant`.

reentrant functions

You can selectively define functions as reentrant (`_reentrant` keyword). Reentrant functions can be invoked recursively. Interrupt programs can also call reentrant functions.

register bank

Each function may contain a specification regarding the register bank to be used (`_using` keyword).

interrupt functions

You can specify interrupt functions directly through interrupt vectors in the C language (`_interrupt` and `__interrupt` keyword). You can also specify the register bank to be used.

3.2 ACCESSING MEMORY

cc51 offers two ways of dealing with the separate address spaces of the 8051, which can be combined. You can:

- specify a storage type (and perhaps also a target memory of a pointer) with the declaration of a C variable

and you are able to:

- select a memory model (for a program or per function), specifying which memory space must be used (as default) for all C variables which do not have an explicit storage specifier. This is very useful for compiling existing C source, which does not need to be adapted for the 8051.

In practice the majority of the C code of a complete application is standard C (without using any language extension). You can compile this part of the application without any modification, using the memory model which fits best to the requirements of the system (code density, amount of external RAM etc.).

Only a small part of the application uses language extensions. These parts often deal with items such as:

- I/O, using the special function registers
- high execution speed needed
- high code density needed
- access to non–default memory required (ROM, internal RAM)
- bit type needed
- C interrupt functions

3.2.1 STORAGE TYPES

cc51 supports the architecture of the 8051 microprocessors and all 8051 derivatives completely. It has full access to all hardware components of the 8051. An object other than a function or an automatic (stack) variable cannot be referred to solely by its starting address, because this might be valid for several address spaces. You can explicitly assign each variable to one of the address spaces (`data`, `bdat`, `idat`, `pdat`, `xdat`, `rom`) by using a type specifier. This specifier determines the 'storage type' of static objects.

Accessing the internal data memory (`_data`, `_idat`) is considerably faster than accessing the external data memory (`_xdat`). Therefore, it is useful to place often used variables into internal data memory, and to place larger and less often referenced data elements into the external data memory.

cc51 recognizes the following storage type specifiers:

Storage Type	Description
_data / data	direct addressable on chip RAM
_bdat	bitaddressable on chip RAM
_idat / idat	indirect addressable on chip RAM
_pdat / pdat	external RAM within 256 bytes page
_xdat / xdat	external RAM
_rom / rom	internal/external ROM

Table 3–1: Storage type specifiers

cc51 treats the storage specifier `_rom` type in a special way: it always implies the type qualifier `const`.

Const Qualifier

The ANSI standard states that the type qualifier `const` can be used to specify 'read–only' objects (or: are not 'lvalues'). An ANSI C compiler may allocate static `const` objects in ROM memory. However, since ROM is a different memory space (which needs special instructions to access), this is not possible for a 8051 C compiler. On the other hand, **cc51** treats `_rom` variables as if declared with a `const` qualifier. So, **cc51** treats `const` just as a type qualifier, which allows the compiler to check on illegal lvalue use. This is exactly the way it is meant to be used in the ANSI definition.

Example:

```
func( i )
const int i;
{
  i++; /* results in error message from cc51 */
}
```

So `const` is no storage type specifier, but a type qualifier (like `unsigned` and `volatile`).

Examples using explicit storage types:

```
_data char  c;
rom   char  text[] = "No smoking";
_xdat int   array[10][4];
idat  long  l;
_pdat int   i;
```

allocating:

1	byte in direct addressable on chip RAM for c
11	bytes in ROM for the initialized character array text[]
80	bytes in external RAM for array
4	bytes in indirectly addressable on chip RAM for l
2	bytes in one page of external RAM for i

The memory type specifiers are treated like any other type specifier (e.g. `unsigned`). This means the examples above can also be declared (exactly the same):

```
char data  c;
char _rom  text[] = "No smoking";
int  xdat  array[10][4];
long _idat l;
int  pdat  i;
```

An object must be fully contained in a single storage section. See section 3.19, *Structure Tags*, for details.

3.2.2 MEMORY MODELS

cc51 has four memory models: small, auxpage, large and reentrant. You can select one of these models with the **–M** option. Each model uses a different default storage type for (non–register) automatic variables, (non–register) parameter passing areas and declarations without an explicit storage type. Parameter passing in the SMALL model is performed in internal data memory. The AUXPAGE and LARGE model permit parameter passing in external memory. The REENTRANT model permits parameter passing via a virtual software stack in external memory.

cc51 also supports mixed memory models; for example, a program created in the large memory model can be accelerated, in which functions are partially distributed to the small model.

The following table gives an overview of the different memory models.

Memory Model	Non–register Parameters / automatics	Other C variables	Max RAM size	Default storage type
small	direct addressable internal RAM	direct addressable internal RAM	128	data
auxpage	one page of external RAM	one page of external RAM	256	pdat
large	external RAM	external RAM	64k	xdat
reentrant	virtual stack in external RAM	external RAM	64k	xdat

Table 3–2: Memory models

Parameters can also be passed in registers, see section 3.4, *Function Parameters*.

Automatics and parameters may be placed in registers. See section 3.7, *Register Variables*. Automatics in the auxpage and large memory models may be placed in internal RAM also. See section 3.6, *Automatic Variables* in this chapter.

Function return addresses are on the real stack. In the small memory model all objects as well as the stack, must fit in the internal RAM. The stack length is critical since its real length depends upon the nesting depth of the various functions.

The auxpage model is especially interesting for derivatives supporting 256 bytes of 'external' RAM on chip. With these derivatives, P2 can be used to specify the external RAM page to be used for paged data (`_pdat`). With other 8051 derivatives, P2 must be set to 0, and cannot be used for other purposes in this model.

Each of the memory models has advantages and disadvantages, especially concerning the access efficiency and the length of the address space. Therefore, **cc51** allows you to mix models.

3.2.2.1 MIXED MEMORY MODEL PROGRAMMING

It is possible to specify a memory model on one function. You can use one of the following keywords:

_small	small model
_aux	auxpage model
_large	large model
_reentrant	reentrant model

When you use reentrant functions, a virtual stack is needed. You also need to change your start–up code. A reentrant function can be called recursively; in addition, calls are allowed at any time, even from interrupt functions.

Example

Suppose the default memory model is large (option **–Ml**). In this case all functions are defined `_large` by default. You may, however, overrule this default behavior by specifying one of the other memory function qualifiers. In this example the `_small` function qualifier is used to obtain fast (direct) data access. All function parameters and local data are stored in internal RAM.

```
/* -Ml: Default is large model */

int
diff( int first, int second )
{    /* large model */
     return( first - second );
}

int _small
sum( int first, int second )
{    /* small model */
     return( first + second );
}
```

3.2.2.2 _MODEL AND _ROMMODEL

cc51 introduces the predefined preprocessor symbols _MODEL and _ROMMODEL. The value of _MODEL represents the memory model selected (**–M** option). The value of _ROMMODEL represents the rom model selected (**–r** option). These can be very helpful in making conditional C code in one source module, used for different applications in different memory models. See also section 3.22, *Portable C Code*, explaining the include file `cc51.h`.

The value of _MODEL is:

small model	's'
auxpage model	'a'
large model	'l'
reentrant model	'r'

The value of _ROMMODEL is:

small	's'
medium	'm'
large	'l'

Example:

```
#if _MODEL == 'a' || _MODEL == 'l' /* non-small
                                    * static model
                                    */
...

#endif

#if _ROMMODEL == 'l'               /* large rom model */
...

#endif
```

3.2.3 THE _AT() ATTRIBUTE

In C–51 it is possible to place certain variables at absolute addresses. Instead of writing a piece of assembly code, a variable can be placed on an absolute address using the `_at()` attribute.

Example:

```
_xdat unsigned char Display _at( 0x2000 );
```

The example above creates a variable with the name `Display` at address 0x2000 in external RAM. In the generated assembly code an absolute section will appear like 'XSEG AT 2000H', on this position space is reserved for the variable `Display`.

A number of restrictions are in effect when placing variables on an absolute address:

- Only global variables can be placed on absolute addresses. Parameters of functions, or automatics within functions cannot be placed on an absolute address.
- When declared 'extern', the variable is not allocated by the compiler. When the same variable is allocated within another module but on a different address, the compiler, assembler or linker will not notice.
- When the variable is declared 'static', no public symbol will be generated (normal C behavior).
- Absolute variables cannot be initialized, except for absolute variables declared in rom.
- Functions cannot be declared absolute.
- Absolute variables cannot overlap each other, declaring two absolute variables on the same address will cause an error generated by the assembler or by the linker. The compiler does not check this.
- Declaring the same absolute variable within two modules will also produce conflicts during link time (except when one of the modules declares the variable 'extern').

3.2.4 THE _ATBIT() ATTRIBUTE

In C–51 it is possible to define bit variables within a **_bitbyte** or (bitaddressable) **_sfrbyte** variable. This can be done with the **_atbit()** attribute. The syntax is:

```
_atbit( bytename, offset )
```

where, *bytename* is the name of a **_bitbyte** or **_sfrbyte** variable and *offset* is the bit–offset with the variable.

Examples:

```
_sfrbyte SCON = 0x98;
_sfrbit  SM1  _atbit( SCON, 6 );

_bitbyte bv;   /* bitaddressable byte */
_bit     myb  _atbit( bv, 3 );

if ( myb )     /* is the same as specifying */
     myb = 0;

if ( _getbit( bv, 3 ) )   /* same code generated */
     _putbit( 0, bv, 3 );
```

The first example defines an sfrbit within an sfrbyte. The second example defines a bitaddress within a bitaddressable byte. For more information on SFR variables see section 3.3.6, *Special Function Registers*. For more information on **_bitbyte** variables see section 3.3.5, *The _bitbyte Type*.

3.3 DATA TYPES

All ANSI C data types are supported, except double and long double, which both are evaluated as floats. In addition to these types, the `_sfrbit`, `_sfrbyte`, `_sfrword`, `_bit` and `_bitbyte` types are added. Two types of pointers are recognized. Object size and ranges:

Data Type	Size (in bytes)	Range
_bit / bit	1 bit	0 or 1
_sfrbit	1 bit	0 or 1
signed char	1	–128 to +127
unsigned char	1	0 to 255
_sfrbyte	1	0 to 255
_bitbyte	1	0 to 255 (byte in bitaddressable RAM)
_sfrword	2	0 to 65535
signed short	2	–32768 to +32767
unsigned short	2	0 to 65535
signed int	2	–32768 to +32767
unsigned int	2	0 to 65535
enum	2	0 to 65535
signed long	4	–2147483648 to +2147483647
unsigned long	4	0 to 4294967295
float	4	+/– 1.176E–38 to +/– 3.402E+38
1–byte pointer (pointer to data, idat or pdat)	1	0 to 255
2–byte pointer (pointer to xdat or rom)	2	0 to 65535

Table 3–3: Data types

– `_bit`, `_sfrbit`, `char`, `_sfrbyte`, `_bitbyte`, `_sfrword`, `short`, `int` and `long` are all integral types, supporting all implicit (automatic) conversions.

- **cc51** generates instructions using (8 bit) character arithmetic, when it is correct to evaluate a character expression this way. This results in a higher code density compared with integer arithmetic. A special section *Character Arithmetic* provides details.
- the 8051 convention is used, storing variables with the most significant part at the lower memory address (Big Endian).
- float is implemented in big endian IEEE 32–bit single precision format.
- with the **–se** option small enumeration types can be treated as `char` instead of `int`.

3.3.1 SIGNED CHARACTERS

The character type is treated as `signed char` by default. You can overrule this default with the **–u** command line option, which sets the default to unsigned char.

Examples:

The following declarations are idential when **–u** is not used.

```
char c;
signed char c;
```

The following declarations are idential when **–u** is used.

```
char c;
unsigned char c;
```

3.3.2 ANSI C TYPE CONVERSIONS

According to the ANSI C X3.159–1989 standard, a character, a short integer, an integer bit field (either signed or unsigned), or an object of enumeration type, can be used in an expression wherever an integer can be used. If a `signed int` can represent all the values of the original type, then the value is converted to `signed int`; otherwise the value will be converted to `unsigned int`. This process is called *integral promotion*.

Integral promotion is also performed on function pointers and function parameters of integral types using the old–style declaration. To avoid problems with implicit type conversions, you are advised to use function prototypes.

Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*.

Integral promotions are performed on both operands; then, if either operand is `unsigned long`, the other is converted to `unsigned long`.
Otherwise, if one operand is `long` and the other is `unsigned int`, the effect depends on whether a `long` can represent all values of an `unsigned int`; if so, the `unsigned int` operand is converted to `long`; if not, both are converted to `unsigned long`.
Otherwise, if one operand is `long`, the other is converted to `long`.
Otherwise, if either operand is `unsigned int`, the other is converted to `unsigned int`.
Otherwise, both operands have type `int`.

See also section 3.3.3, *Character Arithmetic*.

Sometimes surprising results may occur, for example when unsigned char is promoted to int. You can always use explicit casting to obtain the type required. The following example makes this clear:

```
static unsigned char a=0xFF, b, c;

void f()
{
     b=~a;
     if ( b == ~a )
     {
          /* This code is never reached because,
           * 0x0000 is compared to 0xFF00.
           * The compiler converts character 'a' to
           * an int before applying the ~ operator
           */
          ...
     }
```

```
        c=a+1;
        while( c != a+1 )
        {
             /* This loop never stops because,
              * 0x0000 is compared to 0x0100.
              * The compiler evaluates 'a+1' as an
              * integer expression. As a side effect,
              * the comparison will also be an integer
              * operation
              */
        ...
        }
}
```

To overcome this 'unwanted' behavior use an explicit cast:

```
static unsigned char a=0xFF, b, c;

void f()
{
        b=~a;
        if ( b == (unsigned char)~a )
        {
             /* This code is always reached */
             ...
        }

        c=a+1;
        while( c != (unsigned char)(a+1) )
        {
             /* This code is never reached */
        ...
        }
}
```

Keep in mind that the arithmetic conversions apply to multiplications also:

```
static int      h, i, j;
static long     k, l, m;

/* In C the following rules apply:
 *        int  * int          result: int
 *        long * long         result: long
 *
 *  and NOT    int * int      result: long
 */
```

```
void f()
{
     h = i * j;           /* int  * int  = int      */
     k = l * m;           /* long * long = long     */

     l = i * j;           /* int * int = int,
                           * afterwards promoted (sign
                           * or zero extended) to long
                           */
     l = (long) i * j;    /* long * long = long     */
     l = (long)(i * j);   /* int * int = int,
                           * afterwards casted to long
                           */
}
```

3.3.3 CHARACTER ARITHMETIC

cc51 generates code using 8 bit (character) arithmetic as long as the result of the expression is exactly the same as if it was evaluated in integer arithmetic. This must be done, because ANSI does not know character arithmetic and character constants. Because the 8051 is an 8 bit microcontroller, **cc51** tries to use the 8 bit instructions for character arithmetic as much as possible. If not possible, 16 bit arithmetic is used. So it is recommended to use character variables in expressions, because it saves data space for allocation, and often results in a higher code density. You can always force the compiler to use character arithmetic with a character cast.

The following examples clarify when integer arithmetic is used and when character arithmetic:

```
char  a,b,c,d;
int         i;

main()
{
   c = a + b;                 /* character arithmetic */
   i = a + b;                 /* integer arithmetic   */
   i = (char)(a + b);         /* character arithmetic */

   c = a / d;                 /* character arithmetic */
   c = (a + b) / d;           /* integer arithmetic   */
   c = ((char)(a + b)) / d;   /* character arithmetic */

   c = a >> d;                /* character arithmetic */
   c = (a + b) >> d;          /* integer arithmetic   */
```

```
   if ( a > b )         /* character arithmetic */
      c = d;
   if ( (a + b) > c )   /* integer arithmetic   */
      c = d;
}
```

Signed constants between –128 and 127 and unsigned constants between 0 and 255 are treated as a character constant. This means that in:

```
unsigned char c,d;

if ( c > 240 );            /* integer arithmetic   */

if ( d > 5 && d < 240u ); /* character arithmetic */
```

`c` is compared using integer arithmetic because 240 is an integer constant. However, `d` is compared using character arithmetic because both 5 and 240u are character constants.

The following rule applies to hexadecimal and octal constants:
If such a constant fits in an unsigned character it is treated as a character constant. Thus 0xf0 is an unsigned character constant with value 240u.

This rule is derived from a similar approach of the ANSI standard for integer constants, where 0xf000 is treated as an unsigned integer constant.

3.3.4 THE _BIT TYPE

The following rules apply to `_bit` type variables:

1. A _bit type variable is always unsigned.
2. A _bit type variable can be exchanged with all other integral type variables. The compiler generates the correct conversion.

   A _bit type variable is like a boolean. Therefore, converting an int type variable to a _bit type variable does **not** mean the _bit type variable is the least significant bit of the int type variable. It is 1 (true) if the int type variable is not equal to 0, and 0 (false) if the int type variable is 0. In C:

   ```
   bit_variable = int_variable;
   ```

   can be seen as:

   ```
   bit_variable = int_variable ? 1 : 0;
   ```

3. Pointer to _bit and array of _bit are **not** allowed, because the 8051 has no instructions to indirectly access bitaddressable memory.

4. Structure of _bit is supported, with the restriction that no other type than _bit is member of this structure. Structure of _bit is **not** allowed on parameter or return value of a function.

5. A union of a _bit structure and another type is not allowed. The `_bitbyte` type can be used for this purpose.

6. A _bit type variable is **not** allowed as a function parameter of a reentrant function.

7. A _bit type variable is **not** allowed as an automatic variable of a reentrant function. However, a local static _bit variable (within a function) is always allowed.

8. A function may have return type _bit. However, the next rule may not be violated.

9. Evaluation of a complex _bit expression (using non _bit types or _bit return type of a function) is not recursive nor reentrant, because the compiler might need temporary static bit space.

10. A _bit typed expression is not allowed as switch expression.

11. The `sizeof` of a _bit type is 1.

    Normal C bit fields within a structure are not treated like _bit variables, and are therefore not allocated in BIT memory, but in one of the other memory spaces as specified during declaration. Using a structure of _bit type variables results in much better code density, less storage allocation and higher execution speed compared to a structure with a number of 1–bit bit field declarations.

    For example:

```
struct bitvartag {
     /* results in allocation of 3 bits  */
     /* in BIT memory. High code density */
     /* and execution speed              */

     _bit a,
          b,
          c;
} bv;
```

```
struct bitfield {
     /* results in allocation of 16 bits */
     /* in default RAM. Low code density */
     /* and execution speed              */

     unsigned int   a:1,
                    b:1,
                    c:1;
} bf;
```

3.3.5 THE _BITBYTE TYPE

You can declare byte variables in the bit–addressable area as `_bitbyte`. You can access individual bits using the built–in functions `_getbit()` and `_putbit()` or declare the individual bits of this `_bitbyte` variable using `_atbit`. A prototype for these functions is given in the include file `cc51.h`.

For example:

```
_bitbyte bv1, bv2;   /* bitaddressable bytes */

if ( _getbit( bv1, 3 ) )
     _putbit( 1, bv2, 7 );  /* set bit 7 of bv2 */
```

See also section 3.2.4, *The _atbit() Attribute*.

The `_bitbyte` type is subject to the following rules.

1. A _bitbyte type variable is always unsigned.
2. A _bitbyte type variable can be exchanged with all other integral type variables. The compiler generates the correct conversion.
3. Pointer to a _bitbyte variable and array of _bitbyte is allowed.
4. Structure of _bitbyte is supported, with the restriction that no other type than _bitbyte is member of this structure. Structure of _bitbyte is **not** allowed on parameter or return value of a function.
5. A _bitbyte type variable is **not** allowed as a function parameter of a reentrant function.

6. A _bitbyte type variable is **not** allowed as an automatic variable of a reentrant function. However, a local static _bitbyte variable (within a function) is always allowed.

7. A function can **not** have return type _bitbyte.

8. The `sizeof` of a _bitbyte type is 1.

9. A _bitbyte typed expression is allowed as switch expression.

3.3.6 SPECIAL FUNCTION REGISTERS

cc51 allows direct access to all special function registers (bits, bytes and words), as if they were C variables. These special function registers can be used the same way as any other integral data type, including all automatic conversions.

An `_sfrbit` is handled the same way as a `volatile _bit` variable. An `_sfrbyte` is handled as a `volatile unsigned char` variable and an `_sfrword` is handled as a `volatile unsigned int` variable.

In order to 'include' a special function register definition file, you must use the **–C** option. You are able to specify which cpu must be used.

For example, the command:

```
cc51 –C552 i2c.c
```

causes the compiler to look for a file named `reg552.sfr`, and use this file as a special function register definition file. We deliver a number of these definition files with **cc51**, but you can easily make your own one for the 8051 derivative you are using. **cc51** uses the same searching method for these register definition files as with include files. For details, see section 4.3, *Include Files* in chapter *Compiler Use*.

You can also declare sfr–registers within your C–source by using the data types `_sfrbit`, `_sfrbyte` or `_sfrword`. The notation is as follows:

```
_sfrbit   name _atbit( sfrbytename, offset ) ;
_sfrbit   name _at( bitaddress ) ;
_sfrbyte  name _at( byteaddress ) ;
_sfrword  name _at( byteaddress ) ;
```

where, *name* must be replaced with the name of the sfr–register you want to specify. *bitaddress/byteaddress* is the bit or byte address of the sfr–register. *offset* is the bit–offset in an sfrbyte.

Because these registers are placed in the sfr–area of the processor, the compiler will not allocate any storage space.

`_sfrword` allows access to 16–bit SFRs defined on consecutive SFR byte addresses. For example, given two SFR bytes RCAP2L and RCAP2H on addresses 0xCA and 0xCB, these can be accessed simultaneously using `_sfrword RCAP2`. Note should be taken on the order of the separate SFR bytes. By default, the compiler treats all integer types according to big endianness, which means that the high byte is expected on the lower address. In case of RCAP2 however, the high byte RCAP2H is defined on the higher address, hence it should be treated little endian. To support this case you can use the specifier `_little` on the `_sfrword` definition like:

```
_sfrword _little RCAP2 _at(0xCA);
```

You can use the specifier `_little` only on `_sfrword` types.

The words 'sfrbyte', 'sfrword' and 'sfrbit' are not reserved words for **cc51**. So, you can use these words as identifiers. **cc51** does not generate symbolic debugging information for special function registers, because they are already known by the debugger.

Appendix B shows the contents of one of the register definition files delivered with this package (`reg51.sfr`).

Because the special function registers are dealing with I/O, it is not correct to optimize away the access to them. Therefore, **cc51** deals with the special function registers as if they were declared with the `volatile` qualifier.

For example:

```
int               i;
volatile int      v;

main()
{
      i;    /* optimized away                                */
      SBUF; /* access SBUF register (implicit volatile)     */
      v;    /* volatile: access variable                    */
}
```

3.4 FUNCTION PARAMETERS

cc51 supports (ANSI) prototyping of function parameters. Therefore, **cc51** allows passing parameters of type char, and (in static models only) of type `_bit`, **without** converting these parameters to `int` type. This results into higher code density, higher execution speed and less RAM data space needed for parameter passing. This is very important in single chip applications.

For example, in the following C code:

```
void func( char number, _bit status, long value ); int
printf( char *format, ... );

void
main(void)
{
     int  i;
     char c;
     _bit b;

     func( c, b, i );
     printf( "c=%d, b=%d, i=%d\n", c, b, i );
}
```

the code generator uses the prototype of `func()` and:

- passes `c` as a byte
- passes `b` as a _bit (in bit–memory)
- promotes `i` to long before passing it as a long

However, the code generator does not know anything of the `printf()` arguments, because this function is declared with a variable argument list. If there is no prototype (as with the old style K & R functions), the compiler promotes both char type and _bit type parameters to int type, the same way an automatic conversion is done in an assignment of a char/_bit type variable to an int type variable. So, with the `printf()` call the code generator:

- promotes `c` to int before passing it as int
- promotes `b` to int before passing it as int
- passes `i` as int

A lot of execution time of an application is spent transferring parameters between functions. Therefore this is an area which is very interesting for optimizations. **cc51** has several different ways of parameter passing, which depend on the memory model/function qualifier used and on the parameter passing method selected. The memory model can be specified with the command line option **–M{s|a|l|r}**. A memory model on a function can be specified with one of the function qualifiers `_small`, `_aux`, `_large` or `_reentrant`.

The fastest parameter transport is via registers. Therefore, the compiler by default treats functions as `_regparm` functions. Up to three parameters can be passed via CPU registers. If a register is no longer available for a parameter or if a function is specified as `_cdecl`, parameter passing occurs in the fixed memory areas; the address space which is used for parameter passing is dependent on the memory model or function qualifier used.

If functions and function prototypes are not explicitly programmed with `_regparm` or `_cdecl`, the parameter passing method selected depends on the option **–Or**/**–OR**. Option **–Or** treats those functions as `_regparm` functions (default), and option **–OR** treats those functions as `_cdecl` functions.

1. `_regparm _small`
   first parameters in registers, next in static area using naming convention

2. `_cdecl _small`
   in static area using naming convention

3. `_regparm _auxpage`
   first parameters in registers, next in static area using naming convention

4. `_cdecl _auxpage`
   first parameters in fast area, next in static area using naming convention

5. `_regparm _large`
   first parameters in registers, next in static area using naming convention

6. `_cdecl _large`
   first parameters in fast area, next in static area using naming convention

7. `_regparm _reentrant`
   first parameters in registers, next via virtual stack

8. `_cdecl _reentrant`
   via a virtual stack

   Up to three parameters can be passed via CPU registers. The following table shows how arguments are passed via the register parameter passing protocol.

Parameter	char	_data/ _idat / _pdat pointer	int	_xdat/ _rom pointer	long	float
parm1	r7	r7	r67	r67	r4567	r4567
parm2	r5	r5	r45	r45	–	–
parm3	r3	r3	r23	r23	–	–

Table 3–4: Register usage for parameter passing

After a long or float type argument, one more argument can be placed in r3/r23. Structures, unions and bits are never passed via registers.

The following examples clarify the parameter passing conventions:

Example with one argument:

```
func1( int a )
```

– `a` is the first parameter and is passed in registers r6/r7.

Example with three arguments:

```
func2( int b, int c, int _xdat *d )
```

– `b` (first parameter) is passed in registers r6/r7.
– `c` (second parameter) is passed in registers r4/r5.
– `d` (third parameter) is passed in registers r2/r3.

Example with two long/float arguments:

```
func3( long e, long f )
```

– `e` (first parameter) is passed in registers r4/r5/r6/r7.
– `f` (second parameter) cannot be passed through registers anymore; parameter is passed in static area using naming convention.

Example with one long/float and one other argument:

```
func4( float g, char h )
```

– `g` (first parameter) is passed in registers r4/r5/r6/r7.
– `h` (second parameter) is passed in registers r3 (see the note above).

For auxpage/large functions, parameters passed as _cdecl functions result in faster parameter transport, using a static 'fast internal RAM' area for each register bank. This fast internal RAM is named __PARM*x* (*x* stands for the register bank used). Because it is very important to optimize parameter passing, parameters are placed in this area (4 bytes). Very often the parameter computation can be done directly into this area. The rest of the parameters are passed the conventional way: via the static area in external/PDAT memory of the function, using a naming convention.

All arguments which do not fit in the registers are passed in the same manner as for _cdecl declared functions. There is one exception to this rule, the fast parameter area is never used for _regparm functions.

The optimization for _cdecl functions is only done if the C program contains valid prototype declarations of the called functions and their parameters and the called functions do **not** have a variable argument list (ANSI notation of prototype declaration, using three dots, e.g.: `void f(char *, ...);`. Therefore, it is very important to use function prototypes and new style declarations, because this results in faster code.

A function that does not call any other function is called a 'leaf' function. If a function is a leaf function and the C code does not calculate the address of a parameter (via the & operator), the parameters of this function do not have to be copied to the static function parameter area. Thus, the parameters of such a function are left in fast internal RAM.

Non–leaf functions must copy the parameter registers in the static function parameter area at entry, as if they were placed there by the caller.

Note that run–time errors may appear if a C function is called using a valid prototype, while the declaration of the C function itself is **not** using this prototype (and vice versa).

For non–reentrant functions the parameter area of a function is allocated static too. This introduces a limit for functions with a variable argument list. Allocation is done by the compiler during the function definition. You are able to specify the size (in bytes) which must be allocated by the compiler for a variable argument list with the –**a***size* option. The default is 20 bytes.

For example, in a single chip application using `printf()` 20 bytes is probably far too much. Therefore the `printf()` function is delivered in C source and can be recompiled using the **–a** option, and replaced in the library. There may also be another reason to replace the `printf()` function in the library, but this is explained in section 3.10, *Strings*.

Example:

replacing `printf()` of small library with a version using less RAM data as parameter area:

```
cc51  –Ms –a10 _printf.c
asm51 _printf.src noprint nodebug
ar51  crv c51s.lib _printf.obj
```

How to use the library manager (**ar51**) is described in the *8051 Cross–Assembler, Linker, Utilities User's Manual*.

The other functions having a variable argument list (`sprintf()`, `scanf()` and `sscanf()`) are also delivered in C source.

Of course, a replacement as described above is likely to be done in small static models only, because RAM data is very scarce in this model. The reentrant model does not use static memory for parameter passing, so you never have to use the –**a** option with this model. Because parameters are passed on a (software) stack in external RAM, the only thing you have to do with this model is to be sure the system has enough stack space. With the auxpage/large (static) models the –**a** option applies, although more RAM space is available and there is less need to use it. Of course, when you use memory models on functions, using the `_small`, `_aux` or `_large` function specifier, the **–a** option can be useful again.

3.5 FUNCTION OVERLAY

When you use non–reentrant functions, overlaying is default done by **cc51**. **cc51** allocates all automatics and function parameters in overlayable (BIT and DATA) segments.

link51 is capable of overlaying these segments with overlayable segments of other functions, when these functions have no (calling) reference to each other. Note that the function overlay control (**FO**) must be specified to **link51**. The overlaying mechanism deals with all RAM segments (BIT, DATA, IDAT, PDAT and XDAT).

3.6 AUTOMATIC VARIABLES

In non–reentrant functions recursion is not possible. In these functions automatic variables are not allocated on a stack, but in a static area. In a reentrant function automatic variables are treated the conventional way: passed via the stack. In static functions it is possible to force an automatic to a specified memory by using a storage type specifier. The automatics are still overlayable with automatics of other functions. Automatics are subject to the memory model selected. In a static model this means static allocation in one of the RAM memory spaces. In the reentrant model this means dynamic allocation on the stack.

Although automatic variables are allocated in a static area with non–reentrant functions, they are **not** the same as local variables (within a function) which are declared to be static by means of the `static` keyword.

The difference is:

- (as in the 'normal' approach) it is not guaranteed that an automatic variable still has the same value as the previous time the function returned, because it may have been overlaid with another automatic variable of another module.
- (as in the 'normal' approach) it is guaranteed that the value of the static variable is the same as the previous time the function returned. Static variables are never overlaid.

To generate code which is as fast as possible, **cc51** tries to place some automatic variables which are used the most into internal RAM, so these variables are treated as 'register variables' (for auxpage and large functions only). For leaf functions, automatic variables can be placed in registers also. See section 3.7, *Register Variables*.

cc51 does this only to variables which could be defined `register` by the programmer himself too (i.e. no address is taken from this variable).

The maximum amount of internal RAM that may be used for such local variables can be changed with the **–x***size* option or with the **extend** *size* pragma. The default size is 4 bytes. Note that these bytes can be allocated for each function. These bytes are overlayable.

When some variables are defined register in the C source, these variables are always placed in internal RAM and they reduce the number of automatics placed in internal RAM (see the example below).

The option FUNCTIONOVERLAY should be passed to the linker, otherwise you get the message "ADDRESS SPACE OVERFLOW". When you use this option but still get the linker error (SPACE 'DATA' or 'IDATA'), you can try to reduce the usage of internal RAM (use the **–x** option to **cc51**) or use less register/data variables in your program. Note also that the C library is compiled with the **–x4** option (default).

Examples:

All examples assume the 'large' model is used and no objects can be placed in real processor registers.

Example 1:

```
void
func(void)
{
     char c;   /* used 3 times */
     int  i;   /* used 2 times */
     long l;   /* used 4 times */
     ...
}
```

results: 'c' is placed in 'xdat';
'i' is placed in 'xdat';
'l' is placed in 'data'.

Example 2:

```
void
func(void)
{
     char c;      /* used 3 times */
     int  i;      /* used 2 times */
     long l;      /* used 4 times */
     long *p=&l; /* used address of 'l' */
     ...
}
```

results: 'c' is placed in 'data';
'i' is placed in 'data';
'l' is placed in 'xdat'.
'l' cannot be made a register variable because the address of it is used.

Example 3:

```
void
func(void)
{
     register char c;      /* used 3 times */
     int  i;               /* used 2 times */
     long l;               /* used 4 times */
     ...
}
```

results: 'c' is placed in 'data', declared register;
'i' is placed in 'data', fits in register area next to 'c';
'l' is placed in 'xdat', not enough space in register area;

Example 4:

```
void
func(void)
{
     register char  c;
     register int   i;
     register long  l;
     ...
}
```

results: 'c', 'i' and 'l' are placed in 'data'. All variables called `register` are always placed in 'data'. Note that this routine takes 7 bytes of 'data' space, while the previous examples do not exceed the space defined by the **-x** option.

3.7 REGISTER VARIABLES

In C the `register` type qualifier tells the compiler that the variable will be used very often. So the code generator must try to reserve a register for this variable and use this register instead of the data location of this automatic variable. The 8051 has only eight registers (R0–R7, all eight bit), which are also needed by the code generator for normal code generation (indirection, intermediate results etc.). When possible, the compiler tries to allocate some automatic objects or parameter objects within these registers (or the B register). **cc51** offers you the following implementation of register variables for the different memory models:

all models: The compiler tries to place parameters and automatic variables with the processor registers (R0–R7 and B). This is only done for functions not calling other functions (leaf functions). When allocating objects in registers, the code generator gives preference to objects declared with the `register` keyword. For every object not placed in registers, the next rules apply.

small: In this model automatic variables are allocated in `data`, which is directly addressable on–chip RAM. This memory is accessed by the 8051, the same way as certain registers (e.g. B, DPL, DPH, pushing/popping of registers: ACC, AR0– AR7). The code using this `data` memory has a very high execution speed, so in this model there is no need to treat a register variable in a special way, because all automatic variables are accessed with a speed comparable to a real register.

auxpage/large:
In order to increase execution speed and code density of register variables the `register` keyword causes the variable to be placed in fast internal `data`. These variables are overlayable with register variables and `data` automatics of other functions. Therefore, this optimization is default on with all static models. When no register variables are used, the compiler tries to use some automatic variables as register variables. See section 3.6, *Automatic Variables*.

reentrant: Register variables are recognized by the compiler and treated in a special way:

- a special area of 8 bytes in `data` memory is reserved by the compiler for register variables.
- this introduces a maximum of register variables: four integers or far pointers, eight characters, two longs or some combination.

3.8 INITIALIZED VARIABLES

Non automatic initialized variables use the same amount of space in both ROM and RAM (for all possible RAM memory spaces). This is because the initializers are stored in ROM and copied to RAM at start–up. This is completely transparent to the user. The only exception is an initialized variable residing in ROM, by means of the `_rom` storage type specifier.

Examples *(large memory model) :*

```
int         i = 100;        /* 2 bytes in ROM and
                               2 bytes in XDAT    */
_rom int    j = 3;          /* 2 bytes in ROM     */
char       *p = "TEXT";     /* 7 bytes in ROM and
                               7 bytes in XDAT:
                               2 bytes for p,
                               5 bytes for "TEXT" */
_rom  char[] = "HELP";      /* 5 bytes in ROM     */
_data char c = 'a';         /* 1 byte in ROM and
                               1 byte in DATA     */
```

3.9 TYPE QUALIFIER VOLATILE

You can use the `volatile` type qualifier when modifications on the object have undesired side effects when they are performed in the regular way. It may be undesired that the compiler attempts to optimize a memory update by keeping the value in a register (e.g., a hardware register). When a variable is declared with the `volatile` qualifier, the compiler disables such optimizations. The ANSI report describes that the updates of volatile objects follow the rules of the abstract machine (the target processor) and thus access to a volatile object becomes implementation defined.

Example:

```
const volatile int real_time_clock _at(0x1200);

/*   define the real time clock register;
     it is read-only (const);
     read operations must access the real memory
     location (volatile)
*/
```

3.10 STRINGS

In this section the word 'strings' means the separate occurrence of a string in a C program. So, array variables initialized with strings are just initialized character arrays, which can be allocated in any memory type, and are not considered as 'strings'. See section 3.8, *Initialized Variables*, for more information on this topic.

cc51 places strings in both ROM and RAM. Where strings in RAM are placed depends on the specified memory model. If the –**S** option is used, the compiler places all strings in ROM only.

Library routines containing pointer arguments always expect the target memory of these pointers to be the default RAM of the memory model used to make this library.

For example:

```
int printf( const char *format, ... );
```

In large memory model, this means `printf()` expects the address of the format string (the first argument) to have memory type `_xdat`. Therefore, the C startup code of the large memory model copies all strings from ROM to XDAT. So, the statement:

```
printf( "Hello world\n" );
```

is executed correctly, because **cc51** passes the address of the allocated XDAT area (filled at C startup time) to `printf()`.

However, when using a microcontroller in a single chip application, you must be able to allocate strings in ROM only, and adapt your C source code to access these strings. The next example shows how strings can be placed in ROM only:

```
rom char hello[] = "Hello\n"; /* initialized array
                                in ROM only */
char *world = "world\n";      /* initialized pointer
                                to string in XDAT */
```

With the –**S** option on the command line:

```
rom char hello[] = "Hello\n"; /* initialized array
                                in ROM only */
rom char *world = "world\n";  /* initialized pointer
                                to string in ROM */
```

The definition of pointer 'world' should change because it is pointing to ROM now instead of external RAM.

The third method is to use '#pragma romstring', for example:

```
#pragma romstring
rom char hello[] = "Hello\n"; /* initialized array
                                 in ROM only */
rom char *world = "world\n";  /* initialized pointer
                                 to string in ROM */
```

See section 4.4, *Pragmas* in chapter *Compiler Use*, for more information about the pragmas `romstring` and `ramstring`.

These ROM typed strings can be accessed via a pointer to rom, so C functions can be made to manipulate (copy, print) these strings in a user friendly way. The standard library contains a number of memory copy/move functions from and to non default memory, which all are defined in the header file `string.h`.

Because standard library functions expect addresses in RAM only, these strings cannot be passed to these functions. Therefore we deliver the `printf()` function in C source, so you can recompile it, using a pointer to ROM as format string. The only thing you have to do is to define the following preprocessor symbol:

```
#define FORM_CONST
```

in both the `_printf.c` module and the header file stdio.h. How to compile the `_printf.c` module and replace the old `_printf.obj` module in the library has already been described in section 3.4 *Function Parameters*. If FORM_CONST is defined, the following prototype is used in `stdio.h`:

```
int printf( _rom char *format, ... );
```

Be aware, that when using this version of `printf()`, all modules calling `printf()` must have target memory ROM as first argument. So when strings are used, the **–S** option must be on (or pragma romstring must be used). All other arguments of `printf()` always remain in the default RAM memory of the memory model used.

Everything explained above for `printf()` also applies to: `fprintf()`, `sprintf()`, `vprintf()`, `vfprintf()`, `vsprintf()`, `scanf()`, `fscanf()` and `sscanf()`.

ANSI string concatenation is supported: adjacent strings are concatenated – only when they appear as primary expressions – to a single new one. The result may not be longer than the maximum string length (509 characters).

The Standard states that identical string literals need not be distinct, i.e. may share the same memory. Because memory can be very scarce with microcontroller applications, **cc51** overlays identical strings within the same module.

In section 3.1.4 the Standard states that behavior is undefined if a program attempts to modify a string literal. Because it is a common extension to ANSI (A.6.5.5) that string literals are modifiable, there may be existing C source modifying strings at run–time. This can be done either with pointers, or even worse:

```
"st ing"[2] = 'r';
```

cc51 accepts this statement when strings are in both ROM and RAM. Of course, **cc51** does not allow this statement when the **–S** option is used.

3.11 POINTERS

Some objects have two types: a 'logical' type and a storage type. For example, a function is residing in ROM (storage type), but the logical type is the return type of this function. The most obvious C–51 type having different storage and logical type is a pointer. For example:

```
_rom char *_data p; /* pointer residing in data,
                       pointing to ROM */
```

means `p` has storage type `data` (allocated in on chip RAM), but has logical type 'character in target memory space ROM'. The memory type specifier used left to the '*', specifies the target memory of the pointer, the memory specifier used right to the '*', specifies the storage memory of the pointer.

The memory type specifiers are treated like any other type specifier (like unsigned). This means the pointer above can also be declared (exactly the same) using:

```
char _rom  *_data p;  /* pointer residing in data,
                         pointing to ROM */
```

If the target memory and storage memory of a pointer are not explicitly declared, **cc51** uses the default of the memory model selected. For example, in the large model, the declaration:

```
char        *p;
```

is exactly the same as:

```
_xdat char *_xdat p;
```

cc51 is very efficient in allocating pointers, because it recognizes far (2 byte) and near (1 byte) pointers. Pointers to DATA, IDAT and PDAT have a size of 1 byte, whereas pointers to ROM, XDAT and functions (in ROM) have a size of 2 bytes.

Another example:

```
data struct    {
     int             length;
     char _idat      *p;
} s;
```

The structure '**s**' resides in DATA and the compiler allocates 3 bytes for this structure, because '**s.p**' targets IDAT.

In pointer arithmetic **cc51** checks, besides the type of each pointer, also the target memory of the pointers, which must be the same. For example, it is invalid (and has no use) to assign a pointer to data to a pointer to XDAT. Of course, an appropriate cast corrects the error, but in this case results remain unpredictable.

3.12 FUNCTION POINTERS

In C–51 it is possible to use function pointers. You can pass parameters to indirectly called reentrant functions. You can also pass parameters with function pointers in static memory models as long as the parameters fit in registers.

You can specify the function model of the functions pointed to by a function pointer object. So, in the small memory model it is possible to have function pointers to `_reentrant` functions.

Note that a call to a function via a function pointer is not seen by the linker. When you use one of the static models of C–51, you must specify (in the linker control file) which function calls another function via a function pointer. Without this specification **link51** would overlay data of functions that are indirectly called with other functions (when the FUNCTIONOVERLAY control is specified). This would result in run time errors.

You do not need to specify the function calls to `_reentrant` functions to the linker, because `_reentrant` functions do not allocate overlayable data.

3.13 INLINE C FUNCTIONS

With the **_inline** keyword, a C function can be defined to be inlined by the compiler. An inline function must be defined in the same source file before it is 'called'. When an inline function has to be called in several source files, each file must include the definition of the inline function. This is typically solved by defining the inline function in a header file.

Not using a function which is defined as an **_inline** function does not produce any code. Also during a debug session, the inlined function is not known.

Example (**inline.c**) mixed C and generated code:

```
; inline.c    1 _inline char *mystrcpy( char *s1, const char *s2 )
; inline.c    2 {
; inline.c    3         register char *os1;
; inline.c    4
; inline.c    5         os1 = s1;
; inline.c    6         while ( *s1 = *s2 )
; inline.c    7         {
; inline.c    8                 s1++;
; inline.c    9                 s2++;
; inline.c   10         }
; inline.c   11         return os1;
; inline.c   12 }
; inline.c   13
; inline.c   14 _inline void nop ( int count )
; inline.c   15 {
; inline.c   16         if ( count > 0 )
; inline.c   17         {
; inline.c   18 #pragma asm
; inline.c   19         NOP
; inline.c   20 #pragma endasm
; inline.c   21         nop( count - 1 );
; inline.c   22         }
; inline.c   23 }
; inline.c   24
; inline.c   25 _inline long fib1 ( long n )
; inline.c   26 {
; inline.c   27         return (n < 1 ? 1 : fib1( n - 1 ) + fib1( n - 2 ));
; inline.c   28 }
; inline.c   29
```

```
; inline.c   30 int main ( void )
; inline.c   31 {
        PUBLIC  _?main
INLINE_MAIN_DA  SEGMENT DATA OVERLAY( 0 )
        RSEG    INLINE_MAIN_DA
_10:    DS      100
; $mystrcpy#1$s1 = {R7} (register automatic)
; $mystrcpy#1$s2 = {R6} (register automatic)
; buf = _10 (automatic)
; $mystrcpy$os1 (unused automatic, no space allocated)
INLINE_MAIN_PR  SEGMENT CODE
        RSEG    INLINE_MAIN_PR
_?main:
        USING   0
; inline.c   32         char buf[100];
; inline.c   33
; inline.c   34         mystrcpy( buf, buf+1 );
        MOV     R6,# LOW (_10+1)
        MOV     R7,# LOW (_10)
        SJMP    _9
_8:
        INC     R7
        INC     R6
_9:
        MOV     A,R6
        MOV     R1,A
        MOV     A,R7
        MOV     R0,A
        MOV     A,@R1
        MOV     @R0,A
        JNZ     _8
; inline.c   35
; inline.c   36         nop(0);
; inline.c   37         nop(1);
        NOP
; inline.c   38         nop(3);
        NOP
        NOP
        NOP
; inline.c   39
; inline.c   40         return fib1( 10 );
        MOV     R7,#090H
        MOV     R6,#00H
; inline.c   41 }
        RET
```

The pragmas **asm** and **endasm** are allowed in inline functions. This makes it possible to define inline assembly functions. See also section 3.14, *Inline Assembly* in this chapter.

3.14 INLINE ASSEMBLY

cc51 supports inline assembly using the following pragmas:

#pragma asm	Insert assembly text following this pragma.
#pragma asm_noflush	As #pragma asm, but the peephole optimizer does not flush the code buffer.
#pragma endasm	Switch back to the C language.

C modules containing inline assembly are not portable and are very hard to prototype in other environments.

The peephole optimizer in the compiler maintains a code buffer for optimizing sequences of assembly instructions before they are written in the output file. The compiler does not interpret the text of inline assembly. It passes inline assembly lines directly to the output file. To prevent that instructions in the peephole buffer, which belong to C code before the inline assembly lines, will be written in the output file after the inline assembly text, the compiler flushes the instruction buffer in the peephole optimizer. All instructions in the buffer are written to the output file. If this behavior is not desired the pragma **asm_noflush** starts inline assembly without flushing the code buffer.

See also section 7.9, *Assembly Language Interfacing* in chapter *Run–time Environment*.

3.15 BUILT-IN FUNCTIONS

When you want to use some specific 8051 instructions, that have no equivalence in C, you would be forced to write assembly routines to perform these tasks. However, **cc51** offers a way of handling this in C. Therefore, **cc51** has a number of built–in functions, which are implemented as intrinsic functions:

The names of the built–in functions all have a leading underscore, because the ANSI specification states that public C names starting with an underscore are implementation defined.

Examples of the built–in functions are present in a file called `inline.c`, delivered with the package in the `examples` directory. It shows the way these functions can be used, and the 8051 instructions generated by **cc51**. You can compile the file with the options **–Ms** (small model) and **–s** (mixed C–source).

The following built–in functions are implemented (sample C source with generated assembly are given below):

_testclear

```
bit _testclear( bit semaphore );
```

Read and clear *semaphore* using the JBC instruction.

Returns 0 if *semaphore* was not cleared by the JBC instruction, 1 otherwise.

```
_bit b;
unsigned char   c;

if ( _testclear( b ) ) /* JBC instruction */
        c=1;

... Code ...
     JBC  _b,_5
     SJMP _3
_5:
     MOV  _c,#01H
_3:
```

_da

```
unsigned char _da( unsigned char operand );
```

Decimal adjust *operand* after addition using the ADD and DA instructions.

Returns the result.

```
unsigned char c;

/* decimal adjust after addition */
c = _da( c + 1 );
```

... Code ...

```
     MOV  A,#01H
     ADD  A,_c
     DA   A
     MOV  _c,A
```

_jmp

```
void _jmp( (void)(*)(void) );
```

Perform a jump to the specified function.

Returns nothing.

```
void f( void )
{
}
void g( void )
{
      _jmp(f);
```

... Code ...

```
     JMP _?f
```

_nop

```
void _nop( void );
```

Generate NOP instructions.

Returns nothing.

```
_nop();

... Code ...
      NOP
```

_push

```
void _push( unsigned char address );
```

Push the SFR on the specified *address* on the system stack.

Returns nothing.

```
/* Push the SFR IE and on SFR address 0x89 on
   the system stack */
_push( IE );
_push( 0x89 )

... Code ...
      PUSH 0A8H
      PUSH 089H
```

_pop

```
void _pop( unsigned char address );
```

Pop the SFR on the specified *address* from the system stack.

Returns nothing.

```
/* Pop the SFR IE and on SFR address 0x89 from
   the system stack */
_pop( 0x89 );
_pop( IE );

... Code ...
      POP 089H
      POP 0A8H
```

_rol

```
unsigned char _rol( unsigned char operand,
                    unsigned char count );
```

Use the RL instruction to rotate (left) *operand count* times.

Returns the result.

```
unsigned char c;
int i;
/* rotate left, using int variable */
c = _rol( c, i );

... Code ...
     MOV  R2,_i+1
     INC  R2
     RR   A
_6:
     RL   A
     DJNZ R2,_6
     MOV  _c,A
```

_ror

```
unsigned char _ror( unsigned char operand,
                    unsigned char count );
```

Use the RR instruction to rotate (right) *operand count* times.

Returns the result.

```
unsigned char c;
int i;
/* rotate right, using constant */
c = _ror( c, 2 );
c = _ror( c, 3 );
c = _ror( c, 7 );

... Code ...
     RR   A
     RR   A
     MOV  _c,A
;
     SWAP A
     RL   A
     MOV  _c,A
;
     RL   A
     MOV  _c,A
```

_getbit

```
bit _getbit( _bitbyte operand,
             ICE bitoffset );
```

Returns the bit at *bitoffset* (range 0 – 7) of the bitaddressable *operand* for usage in bit expressions.

ICE denotes that the operand must be an Integral Constant Expression rather than any type of integral expression.

```
_bitbyte bv1;
int i;

if ( _getbit( bv1, 3 ) )
     i = 1;

... Code ...
     JNB  _bv1.3,_4
     MOV  _i,#00H
     MOV  _i+1,#01H
_4:
```

_putbit

```
void _putbit( bit value, _bitbyte operand,
              ICE bitoffset );
```

Assign *value* to the bit at *bitoffset* (range 0 – 7) of the bitaddressable *operand*. ICE denotes that the operand must be an Integral Constant Expression rather than any type of integral expression.

Returns nothing.

```
_bitbyte bv2;

_putbit( 1, bv2, 7 );
_putbit( 0, bv2, 6 );

... Code ...
     SETB _bv2.7
;
     CLR  _bv2.6
```

3.16 INTERRUPT AND USING

A function can be declared to serve as an interrupt service routine (ISR). Interrupt functions cannot return anything and must have a **void** argument type list. Interrupt functions may be implemented directly in C, by using the `_interrupt(`*`interrupt_id`*`)` or `__interrupt(`*`vector_address`*`)` function qualifier. The first one takes an interrupt number as its argument, the second one (double underscore) takes any vector address as its argument. Both function qualifiers can be intermixed.

The relation between the interrupt number and the vector address is: *interrupt_id* = (*vector_address* – 3)/8.

Example with vector address:

```
__interrupt(0x8074) void ISR( void)
{
    return;
}
```

Normally when an interrupt function is called, all registers in the default register bank that are (or could be) used in the interrupt function are saved on the stack so the registers are available for the interrupt routine. After returning from the interrupt routine, the original values are restored from the stack again.

For the 8051 it is possible to assign a new register bank to an interrupt function, which can be used on the processor to minimize the interrupt latency because registers do not need to be pushed on stack. You can switch register banks with the `_using(`*`bank`*`)` function qualifier. For example, in:

```
_interrupt(1) _using(2) void timer(void);
```

An interrupt routine can also handle multiple interrupt numbers. Note that only one `_using()` is allowed. For example, in:

```
_interrupt(1,2,3) _using(2) void isr123(void);
```

or:

```
_interrupt(1) _interrupt(2) _interrupt(3) _using(2)
     void isr123(void);
```

cc51 places a long–jump instruction on the address of the vector of interrupt number 1, to the `timer()` routine, which switches the register bank to bank 2 and saves some more registers. When `timer()` is completed, the extra registers are popped, the bank is switched back to the original value and a RETI instruction is executed.

For more details, see section 7.7, *Interrupt Functions* in chapter *Run–time Environment*.

Because the vector is filled by the compiler (unless disabled by `_interrupt(-1)` or by the –**v** option or by pragma **novector**), the interrupt number must be specified. To find out which interrupt number should be used, see section 7.7, *Interrupt Functions*.

You can call another C function from the interrupt C function. However, this function must be compiled with the same `_using` (register bank) attribute, because **cc51** generates code which uses the addresses of the registers R0–R7. Therefore, the `_using` attribute is also possible with normal C functions (and their prototype declarations). Suppose `timer()` is calling `get_number()`. The function prototype (and definition) of `get_number()` should contain the correct `_using`:

```
_using( 2 ) int get_number( void );
```

cc51 checks if a function calls another function using another register bank, which is an error. The default register bank of a module is 0, or the bank number specified with the –**b** option.

When you want to call a function from within an interrupt function, the called function should have the same `_using` attribute. This has several reasons :

- **cc51** generates fast code, i.e. it may address registers indirectly by use of their addresses in data. Because the register bank is switched, also the register addresses are changed.
- Each function uses a static allocated data space for its parameters and variables. When a function is called from the main C program and by an interrupt function, the values of the variables are overwritten.

For example, the function `display` is declared as :

```
int _cdecl display( char * str );
```

This function uses the area '_display_BYTE' for its parameters. An interrupt routine calling this function immediately overwrites these values. You can solve this problem by calling a second function 'display' when in the interrupt routine. So you must create a function :

```
int _using( 2 ) display2( char *str )
{
     ... code ...
}
```

This function uses its own parameter area called '_display2_BYTE'. The example assumes you use register bank 2 for your interrupt routine. The interrupt routine may now call the created routine 'display2'.

In the reentrant model, register bank switching can be done also. We recommend using a `_large`, `_small` or `_aux` interrupt function and not to call a `_reentrant` function from the interrupt function. In that case you can use the standard library and there is no need to protect. When using interrupts, the same problems occur as described above. Thus the reentrant model provides recursion, but not reentrancy.

However, reentrancy is possible. To get real reentrancy you should not use the `_using()` qualifier. Then the compiler will automatically save all registers at every interrupt.

Besides this, you should not use static variables in your `_reentrant` routines called from the interrupt handler. Using these variables in your routine always overwrites the original contents of it.

In the reentrant model, a software stack pointer is being maintained. Because the instructions needed to update this software stack pointer are divisible, this stack pointer can be in an undefined state at the time of the interrupt. This introduces a problem, when calling another C function from the interrupt C function.

However, if interrupts are (temporarily) disabled, while updating the stack pointer, the problem does not occur. Therefore, we deliver a stack manager module, which disables interrupts during stack pointer updates. This module can replace the original stack manager module in the library. See section 7.10, *Reentrant Model / _reentrant Functions* in chapter *Run–time Environment*, for details on this subject.

Therefore, when using the reentrant model and the standard library, it is not possible to do any stack operations on the virtual stack; that is: access automatics, register variables of `_reentrant` functions or calling another `_reentrant` C function from the interrupt C function.

3.17 REGISTER BANK INDEPENDENT CODE GENERATION

Option **–noregaddr** has been added to the compiler to switch to register bank independent code generation. In order to generate very efficient code the compiler uses absolute register addresses in its code generation. For example a register to register 'move'. Since there is no 'MOV *register*, *register*' instruction, the compiler will generate a 'MOV *register*, *direct*' with the absolute address of the source register as the second operand.

The absolute address of a register depends on the register bank, but sometimes this dependency is undesired. For example when a function is called from both the main thread and an interrupt thread. If both threads use different register banks, they cannot call a function that uses absolute register addresses. To overcome this, the compiler can be instructed to generate a register bank independent function that can be called from both threads.

Example:

```
_noregaddr int func( int x )
{
    /* this function can be called from any function
       indepent of its register bank */
    return x+1;
}
_using(1) void f1( void )
{
    func( 1 );
}

_using(0) void main( void )
{
    func( 0 );
}
```

3.18 C CODE CHECKING: MISRA C

The C programming language is a standard for high level language programming in embedded systems, yet it is considered somewhat unsuitable for programming safety–related applications. Through enhanced code checking and strict enforcement of best practice programming rules, TASKING MISRA C code checking helps you to produce more robust code.

MISRA C specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems. It consists of a set of 127 rules, defined in the document "Guidelines for the Use of the C Language in Vehicle Based Software" published by "Motor Industry Research Association" (MISRA).

Every MISRA C rule is classified as being either 'required' or 'advisory'. *Required* rules are mandatory requirements placed on the programmer. *Advisory* rules are requirements placed on the programmer that should normally be followed. However, they do not have the mandatory status of required rules.

Implementation issues

The MISRA C implementation in the compiler supports most of the 127 rules. Some MISRA C rules address documentation, run–time behavior, or other issues that cannot be checked by static source code inspection. Therefore, some rules are not implemented. These unsupported rules are visible in the **C Compiler | MISRA C | MISRA C Rules** entry of the Project Options dialog in EDE, but cannot be selected (grayed out).

During compilation of the code, violations of the enabled MISRA C rules are indicated with error messages and the build process is halted. For example,

```
E 209: MISRA C rule 9 violation: comments shall not be nested.
```

You can change the level of error messages from errors to **w**arnings on the required MISRA C rules and the advisory MISRA C rules, with the following C compiler command line options:

```
-misrac-required-warnings

-misrac-advisory-warnings
```

Note that not all MISRA C violations will be reported when other errors are detected in the input source. For instance, when there is a syntax error, all semantic checks will be skipped, including some of the MISRA C checks. Also note that some checks cannot be performed when the optimizations are switched off.

Apply MISRA C code checking to your application

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **MISRA C**.

3. Select a MISRA C configuration. Select a predefined configuration for conformance with the required rules in the MISRA C guidelines.

   It is also possible to have a project team work with a MISRA C configuration common to the whole project. In this case the MISRA C configuration can be read from an external settings file.

4. (Optional) In the **MISRA C Rules** entry, specify the individual rules.

From the command line MISRA C can be enabled by the following compiler option:

```
-misracn,n,...
```

where *n* specifies the rule(s) which must be checked.

See Appendix A, *MISRA C* for the supported and unsupported MISRA C rules.

3.19 STRUCTURE TAGS

A tag declaration is intended to specify the lay–out of a structure or union. If a memory type is specified, it is considered to be part of the declarator. A tag name itself, nor its members can be bound to any storage area, although members having type "... pointer to" do require one. A tag may then be used to declare objects of that type, and may allocate them in different memories (if that declaration is in the same scope). The following example illustrates this constraint.

```
struct S {
   _xdat int i;   /* referring to storage: not correct      */
   _idat char *p; /* used to specify target memory: correct */
   };
```

In the example above **cc51** ignores the erroneous `_xdat` storage specifier (without displaying a warning message).

3.20 TYPEDEF

Typedef declarations follow the same scope rules as any declared object. Typedef names may be (re–)declared in inner blocks but not at the parameter level. However, in typedef declarations, memory specifiers are allowed. A typedef declaration should at least contain one type specifier.

Examples:

```
typedef _idat int IDATINT;    /* storage type _idat: OK */
typedef int _data *DATAPTR;   /* logical type _data
                                 storage type 'default' */
```

3.21 SWITCH STATEMENT

cc51 supports three ways of code generation for a switch statement: a jump chain (linear switch), a jump table or a binary search table.

A jump chain is comparable with an if/else–if/else–if/else construction. A jump table is a table filled with JMP instructions for each possible switch value. The switch argument is used as an index to jump within this table. A binary search table is a table filled with a value to compare the switch argument with and a target address to jump to.

By default, the compiler will try to use the switch method which uses the least space in ROM (i.e. table size in ROMDATA plus code to do the indexing).

For a switch with a **long** type argument, only binary search table code is used. For an **int** type argument, a jump table switch is only possible when all case values are in the same 256 value range (i.e. the high byte value of all programmed cases are the same).

It is obvious that, especially for large switch statements, the jump table approach executes faster than the binary search table approach. Also the jump table has a predictable behavior in execution speed. No matter the switch argument, every case is reached in the same execution time.

With a small number of cases, the jump chain method can be faster in execution and shorter in size.

The compiler chosen switch method can be overruled by using:

```
#pragma linear_switch     /* force jump chain code */
#pragma jump_switch       /* force jump table code */
#pragma binary_switch     /* force binary search table
                             code */
#pragma smart_switch      /* let the compiler decide
                             the switch method used */
```

The last one is also the default of the compiler. Using a pragma cannot overrule the restrictions as described earlier.

The `_switch` pragmas must be placed before the function body containing the switch statement. Nested switch statements use the same switch method, unless the nested switch is implemented in a separate function which is preceded by a different `_switch` pragma.

Example

```
/* place pragma before function body */
#pragma jump_switch

void test(unsigned char val)
{ /* function containing the switch */
     switch (val)
     {
          /* use jump table */
     }
}
```

3.22 PORTABLE C CODE

If you are developing C code for the 8051 using **cc51**, you might want to test some code on the host you are working on, using a C compiler for that host. Therefore we deliver the include file `cc51.h`. This header file checks if _CC51 is defined (**cc51** only), and redefines the storage type specifiers if it is not defined.

When using this include file, you are able to use the storage type specifiers (when needed) and yet write 'portable C code'.

Furthermore an adapted prototype of each C–51 built–in function is present, because these functions are not known by another ANSI compiler. If you use these functions, you should write them in C, performing the same job as the 8051 and link these functions with your application for simulation purposes.

3.23 HOW TO PROGRAM SMART IN C-51

If you want to get the best code out of **cc51**, the following guidelines should be kept in mind:

1. If you are using the large model (because it is not possible to use the small model or auxpage model), try to declare the most frequently used variables (both static and automatic) with storage type `data`. If you want your code to remain portable, you can use the `register` keyword. See also section 3.22 *Portable C Code* and section 3.7 *Register Variables*. It is also possible to increase the internal automatics space (**-x** option), so the compiler places more variables in internal RAM.

   Another approach may be even better: always use the small model, so parameter passing is always done via internal RAM. Specify the objects you want to be placed in XDAT.

2. Try to use the `unsigned` qualifier as much as possible (e.g. `for (i = 0; i < 500; i++)`), because unsigned comparisons require less code than signed comparisons.

3. Try to use the smallest data type as possible: bit for boolean usage (flags), character for small loops and so on. See also section 3.3.3, *Character Arithmetic*, and section 3.3.4, *The _bit Type*.

4. If execution speed is important (e.g. interrupt functions and time consuming loops), you must use the **–Of** option or **#pragma optimize f** or **#pragma speed**.

3.24 SOME EXAMPLES OF COMPLEX DECLARATORS

Because the **cc51** has some extensions to support the various memory types of the 8051 processor family, declarations of objects may need some explanation.

First of all, declaration of simple objects is done exactly the same way as in standard C.

For example:

```
char c;
int  i;
long l;
```

When programming portable C–code, declaration of pointers is also standard.

For example:

```
char *pc;
int  *pi;
long *pl;
```

However, for code density it may be desired to place an object in another memory area, this can be done by preceding the object type by the requested data area specifier.

For example:

```
_data char dc;
_xdat int  xi;
_idat long pl;
```

also correct is :

```
char _data dc;
int  _xdat xi;
long _idat pl;
```

Now, pointers to another area than the default (specified by the memory model, see section 3.2.2 *Memory Models*) are declared as follows:

`_data char * pdc;`	Pointer resides in default memory, points to a character in `data`.
`_xdat int  * pxi;`	Pointer resides in default memory, points to an integer in `xdat`.
`_idat long * ppl;`	Pointer resides in default memory, points to a long in `idat`.

Even more difficult, these pointers may be placed in some other data area than the default.

For example:

`_data char * _xdat xpdc;`	Pointer resides in `xdata`, points to a character in `data`.
`_xdat int  * _pdat ppxi;`	Pointer resides in `pdata`, points to an integer in `xdat`.
`_idat long * _idat ippl;`	Pointer resides in `idata`, points to a long in `idat`.

Using objects located in `data` always produce less code than objects in `xdata`. So the smallest code size (and often the fastest execution speed) can be achieved by placing as many objects as possible in `data`. When it is not possible to place all objects in internal RAM, select the objects which are most referenced in the code.

Some examples of complex declarators are given below.

```
_data char c;
_data char * _idat p = &c;
_data char * _idat * pp = &p;
_data char * _idat * * _xdat ppp = &pp;
```

Now `ppp` is a pointer located in `xdat`, points to a pointer in default memory, this points to a pointer in `idat`, which is a pointer to a character in `data`.

```
int _idat * func( void );
int _idat (* _data fp)( void ) = func;
```

Now `func` is a pointer located in `data`, points to a function with no arguments, returning a pointer to an integer in `idat`.

In static memory models it is not possible to call a function indirectly by a function pointer while passing parameters. An indirect call to a function with a void parameter list is still possible.

CHAPTER 4

COMPILER USE

TASKING

4 CHAPTER

4.1 CC51 INVOCATION

The invocation syntax of the C–51 compiler is:

cc51 [[*option*] ... [*file*] ...] ...

The input *file* must have the extension **.c** or **.i**. Options are preceded by a '–' (minus sign). Options cannot be combined after a single '–'. After you have successfully compiled your C sources, the compiler has generated assembly files, with the extension **.src** (the default for **asm51**).

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 **–\?**.

A summary of the options is given below. A more detailed description is given in the next section.

Option	Description
–?	Display invocation syntax
–A[*flag*...]	Enable/disable specific language extensions
–C*cpu*	Use special function register definitions for *cpu*
–D*macro*[=*def*]	Define preprocessor *macro*
–E[**m**]	Preprocess only or emit dependencies
–H*file*	Include *file* before starting compilation
–I*directory*	Look in *directory* for include files
–M{**s**|**a**|**l**|**r**}	Select memory model: small, auxpage, large or reentrant
–O*flag*...	Control optimization
–R*mem*[=*name*]	Change segment name
–S	Put strings in ROM only
–U*macro*	Remove preprocessor *macro*
–V	Display version header only
–a*size*	Change allocated space for variable argument list
–b*number*	Specify default register bank number
–banks	Use bank switch segment name convention
–bp*number*	Enable chip errata bypass

Option	Description
–c*size*	Extend amount of internal RAM to be used as CSE space
–e	Remove output file if compiler errors occur
–err	Send diagnostics to error list file (`.err`)
–f *file*	Read options from *file*
–g[e|f|l|r]...	Enable symbolic debug information
–ivo=*value*	16–bit base address for the interrupt vector table
–**l[i]**	Generate list file; optionally with include files
–m*mem***=***size*	Specify memory *size*
–misrac*n,n,...*	Enable individual MISRA C checks
–misrac–advisory–warnings	Generate warnings for advisory MISRA C rules
–misrac–required–warnings	Generate warnings for required MISRA C rules
–n	Send output to standard output
–nofastparm	Switch off use of fast parameter area
–noregaddr	Get register bank independent code generation
–o *file*	Specify name of output *file*
–pa	Use dual data pointer (Atmel AT8x53, AT89S53, AT89S4D12, AT89S8252)
–pd	Use dual data pointer (Dallas 80C320/520/530, AMD 80C521)
–pp	Use dual data pointer (Philips 51 family)
–ps	Use multiple data pointer (Infineon Technologies C500/C800 series)
–r{s | **m** | **l}**	Select rom model: small, medium or large
–s	Merge C–source code with assembly output
–se	Treat small enumerated types as ’char’ instead of ’int’
–shiftright–signfill	Use sign fill on signed shift right
–t	Display module summary
–u	Treat all ’char’ variables as unsigned
–v	Do not generate interrupt vectors
–vf	Do not generate frame for interrupt handler

Option	Description
–vo	Generate old style interrupt frame
–w[*num*]	Suppress one or all warning messages
–wstrict	Suppress warning messages 183, 196
–x*size*	Extend amount of internal RAM for automatics
–z*pragma*	Identical to '#pragma *pragma*' in the C source

Table 4–1: Compiler options (alphabetical)

Description	Option
Include options	
Read options from *file*	**–f** *file*
Include *file* before starting compilation	**–H***file*
Look in *directory* for include files	**–I***directory*
Preprocess options	
Preprocess only or emit dependencies	**–E**[**m**]
Define preprocessor *macro*	**–D***macro*[=*def*]
Remove preprocessor *macro*	**–U***macro*
Allocation control options	
Put strings in ROM only	**–S**
Change allocated space for variable argument list	**–a***size*
Specify default register bank number	**–b***number*
Extend amount of internal RAM to be used as CSE space	**–c***size*
Specify memory *size*	**–m***mem*=*size*
Extend amount of internal RAM for automatics	**–x***size*
Code generation options	
Use special function register definitions for *cpu*	**–C***cpu*
Select memory model: small, auxpage, large or reentrant	**–M{s|a|l|r}**
Control optimization	**–O***flag*...
Change segment name	**–R***mem*[=*name*]

Description	Option
Use bank switch segment name convention	**–banks**
Enable chip errata bypass	**–bp***number*
16–bit base address for the interrupt vector table	**–ivo=***value*
Switch off use of fast parameter area	**–nofastparm**
Get register bank independent code generation	**–noregaddr**
Use dual data pointer (Atmel AT8x53, AT89S53, AT89S4D12, AT89S8252)	**–pa**
Use dual data pointer (Dallas 80C320/520/530, AMD 80C521)	**–pd**
Use dual data pointer (Philips 51 family)	**–pp**
Use multiple data pointer (Infineon Technologies C500/C800 series)	**–ps**
Select rom model: small, medium or large	**–r**{**s** | **m** | **l**}
Use sign fill on signed shift right	**–shiftright–signfill**
Do not generate interrupt vectors	**–v**
Do not generate frame for interrupt handler	**–vf**
Generate old style interrupt frame	**–vo**
Identical to '#pragma *pragma*' in the C source	**–z***pragma*
Language control options	
Enable/disable specific language extensions	**–A**[*flag*...]
Treat small enumerated types as 'char' instead of 'int'	**–se**
Treat all 'char' variables as unsigned	**–u**
Output file options	
Remove output file if compiler errors occur	**–e**
Send output to standard output	**–n**
Specify name of output *file*	**–o** *file*
Merge C–source code with assembly output	**–s**
Diagnostic options	
Display invocation syntax	**–?**
Display version header only	**–V**
Send diagnostics to error list file (`.err`)	**–err**

Description	Option
Enable symbolic debug information	**–g**[**e**|**f**|**l**|**r**]...
Generate list file; optionally with include files	–**l**[**i**]
Enable individual MISRA C checks	**–misrac***n,n,...*
Generate warnings for advisory MISRA C rules	**–misrac–advisory–warnings**
Generate warnings for required MISRA C rules	**–misrac–required–warnings**
Display module summary	**–t**
Suppress one or all warning messages	**–w**[*num*]
Suppress warning messages 183, 196	**–wstrict**

Table 4–2: Compiler options (functional)

4.2 DETAILED DESCRIPTION OF THE C-51 OPTIONS

Option letters are listed below. Each option (except **–o**; see description of the **–o** option) is applied to every source file. If the same option is used more than once, the first (most left) occurrence is used. The placement of command line options is of no importance except for the **–I** and **–o** options. For those options having a file argument (**–o** and **–f**), the filename may not start immediately after the option. There must be a tab or space in between. All other option arguments must start immediately after the option. Source files are processed in the same order as they appear on the command line (left–to–right).

Some options have an equivalent pragma.

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:

```
cc51  –?
```

-A

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Language Extensions**.
In the **Language extensions** box, select **Enable all language extensions** or select **Custom language extensions** and enable or disable one or more language extensions.

–A[*flags*]

Arguments:

Optionally one or more language extension flags.

Default:

–A1

Description:

Enable/disable language extensions. **–A** without any flags, specifies strict ANSI mode; all language extensions are disabled. This is equivalent with **–ACKLNPSTUVX** and **–A0**.

Flags which are controlled by a letter, can be switched on with the lower case letter and switched off with the uppercase letter. Note that the usage of these options might have effect on code density and code execution performance. The following flags are allowed:

c Default. Perform character arithmetic. **cc51** generates code using 8–bit character arithmetic as long as the result of the expression is exactly the same as if it was evaluated using integer arithmetic. See also section 3.3.3, *Character Arithmetic*.

C Disable character arithmetic.

k Default. Allow keyword language extensions without underscores. For example, both `_xdat` and `xdat` are allowed.

K Only keyword extensions that start with an underscore, such as `_xdat`, are allowed.

l Default. 120 significant characters are allowed in an identifier instead of the minimum ANSI–C translation limit of 31 significant characters. Note: more significant characters are truncated without any notice.

L Conform to the minimum ANSI–C translation limit of 31 significant characters. This makes it possible to translate your code with any ANSI–C conforming C–compiler. Note: more significant characters are truncated without any notice.

n Default. Do not clear non–initialized global variables.

N Non–initialized global variables are cleared at startup.

p Default. Allow C++ style comments in C source code. For example:

```
// e.g this is a C++ comment line.
```

P Do not allow C++ style comments in C source code, to conform to strict ANSI–C.

s Default. __STDC__ is defined as '0'. The decimal constant '0', intended to indicate a non–conforming implementation. When one of the language extensions are enabled __STDC__ should be defined as '0'.

S __STDC__ is defined as '1'. In strict ANSI–C mode (**–A**) __STDC__ is defined as '1'.

t Default. Do not promote old–style function parameters when prototype checking.

T Perform default argument promotions on old–style function parameters for a strict ANSI–C implementation. `char` type arguments are promoted to `int` type and `float` type arguments are then promoted to `double` type.

u Default. Use type `unsigned char` for 0x80–0xff. The type of an unsuffixed octal or hexadecimal constant is the first of the corresponding list in which its value can be represented:

Character arithmetic enabled **–Ac**:

```
char, unsigned char, int, unsigned int, long,
unsigned long
```

Character arithmetic disabled **–AC** (strict ANSI–C):

```
int, unsigned int, long, unsigned long
```

U Do not use type `unsigned char` for 0x80–0xff. The type of an unsuffixed octal or hexadecimal constant is the first of the corresponding list in which its value can be represented:

Character arithmetic enabled **–Ac**:

```
char, int, unsigned int, long, unsigned long
```

Character arithmetic disabled **–AC** (strict ANSI–C):

```
int, unsigned int, long, unsigned long
```

v Allow type cast of an lvalue object with incomplete type `void` and lvalue cast which does not change the type and memory of an lvalue object.

Example:

```
void *p; ((int*)p)++;      /* allowed */
int i; (char)i=2;          /* NOT allowed */
```

V Default. A cast may not yield an lvalue, to conform strict ANSI–C mode.

x Default. Do not check for assignments of a constant string to a non–constant string pointer. With this option the following example produces no warning:

```
char *p;
void main( void ) { p = "hello"; }
```

X Conform to ANSI–C by checking for assignments of a constant string to a non–constant string pointer. The example above produces warning W130: "operands of '=' are pointers to different types".

0 – same as **–ACKLNPSTUVX** (disable all)

1 – same as **–Acklnpstuvx** (default)

Example:

To disable character arithmetic and C++ comments enter:

```
cc51 -ACP test.c
```

-a

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation**. Enter a size in the **Size (in bytes) of the parameter area for non–reentrant functions with a Variable Argument list** field.

–a*size*

Pragma:

#pragma arglist *size*

Arguments:

A number of bytes.

Default:

–a20

Description:

Use *size* for number of bytes to be allocated for function definitions having a variable argument list. The default is 20. This option is applied to the non–reentrant functions only, because reentrant functions use the stack for parameter passing.

Example:

```
cc51 -a10 test.c
```

Pragma `arglist` in the section 4.4, *Pragmas*.

-b

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation**. Enter a register bank number in the **Register bank number (0–3)** field.

–b*number*

Arguments:

A register bank number in the range of 0 to 3.

Default:

–b0

Description:

Select the default register bank value for all functions of the module. Notice no code is generated to switch to this register bank. The default register bank is 0.

Example:

```
cc51 –b1 test.c
```

Section 3.16, *Interrupt and Using*

-banks

Option:

From the **Project** menu, select **Project Options...**
Expand the **Linker** entry and select **Bank Switching**. Specify a **Number of code banks** and enable option **Use alternate bank switch segment name convention**.

–banks

Description:

When you use code bank switching several limitations holds regarding references and calls between segments located in different code banks. In order to easen the linker process you can select a special segment naming convention with the **–banks** compiler option.

It may in some situations be required to still use a different segment name convention, you can use the compiler option **–R** for this.

Example:

To use the bank switch segment name convention, enter:

```
cc51 –banks test.c
```

–R

-bp

Option:

From the **Project** menu, select **Project Options...**
Expand the **Processor** entry and select **Bypasses**. Enable the option **Bypass DS80C390 erratum #6 (DIV AB preceded by ACC access)**.

–bp*number*

Arguments:

TASKING chip erratum bypass number.

Description:

Enable bypass for certain CPU functional problems.

Bypass *number* 1 bypasses DS80C390 erratum #6, and inserts an extra NOP before any DIV AB instruction.

Example:

To bypass DS80C390 erratum #6, enter:

```
cc51 -bp1 test.c
```

See Appendix E, *CPU Functional Problems* for more details.

-c

Option:

From the **Project** menu, select **Project Options...**
Expand the **Processor** entry and select **Processor Selection**. Choose a processor from the list of derivatives or select **User specified CPU** and enter your own processor type.

–C*cpu*

Arguments:

The CPU name which identifies your 8051 derivative.

Description:

Use special function register definitions for *cpu*. The filename looked for is ”reg*cpu*.sfr” in the same way include files whose names are enclosed in ”” are searched.

Example:

To specify to the compiler to look for a file named `regp8xc52.sfr`, and to use this file as a special function register definition file, enter:

```
cc51 –Cp8xc52 test.c
```

Section 3.3.6, *Special Function Registers*, in the previous chapter.

-c

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation**. Enter a size in the **Amount of DATA for optimization (overlayable)** field.

–c*size*

Pragma:

cse *size*

Arguments:

A number of bytes.

Default:

–c0

Description:

With this option you can specify the maximum amount of CSE space which may be used by a function. When no CSEs are found within a function, no space will be allocated for it. CSE space is overlayable. The default *size* is 0. Increase the size value to enable the compiler to allocate some data space for other possible CSE values. When the size is kept '0', the compiler will check for CSE values and tries to place them in register only.

Example:

```
cc51 -c10 test.c
```

Pragma `cse` in the section 4.4, *Pragmas*.

-D

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Preprocessing**. Define a macro (syntax: *macro*[**=***def*]) in the **Define user macros** field. You can specify and 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* 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 compilations. If the command line is getting longer than the limit of the operating system used, the **–f** option is needed.

Example:

The following command defines the symbol `NORAM` as `1` and defines the symbol `PI` as `3.1416`.

```
cc51 –DNORAM –DPI=3.1416 test.c
```

Option **–U**

-E / -Em

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Preprocessing**. Enable the option **Store the C Compiler preprocess output**. Optionally, select one or more of the sub–options.

–E[**m**]

Description:

Run the preprocessor of **cc51** only and send the output to stdout. When you use the –**E** option, use the –**o** option to separate the output from the header produced by the compiler. When you use the **–Em** option, the compiler generates dependency rules which can be used by a 'make' utility.

Examples:

The following command preprocesses the file `test.c` and sends the output to the file `preout`.

```
cc51 –E –o preout test.c
```

The following command generates dependency rules for the file `test.c` which can be used by **mk51** (the 8051 'make' utility).

```
cc51 –Em test.c

test.src :  test.c
```

-e

Option:

EDE always removes the output file on errors.

–e

Description:

Remove the output file when an error has occurred. With this option the 'make' utility always does the proper productions.

Example:

```
cc51 -e test.c
```

-err

Option:

In EDE this option is not so useful. If you would use this option you would not see the error messages in the **Build** tab.

–err

Description:

Write errors to the file *source*.err instead of stderr.

Example:

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

```
cc51 -err test.c
```

–f

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Miscellaneous**. Add the option to the **Additional C Compiler 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.c
```

The command line can now be:

```
cc51 -f mycmds
```

-g

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Debug**. Enable the option **Generate symbolic debug information**. Optionally enable one or more of the other options.

–g[**f**|**e**|**r**|**l**]...

Description:

Add directives to the output files, incorporating symbolic information to facilitate high level debugging.

If **–gf** is used, high level language type information is also emitted for types which are not referenced by variables. Therefore, this sub–option is not recommended.

If **–ge** is used, a NOP instruction is emitted at the start of every C–line. This option can be useful in combination with skidding emulators, executing the instruction which is having a code breakpoint.

If **–gr** is used, two NOP instructions are emitted at the start of every C–line. This option can be useful in combination with a ROM monitor based execution environment, allowing a software breakpoint (3 byte JMP MONITOR) on every C–line. Note that CrossView51 ROM checks and refuses overlapping breakpoints.

If **–gl** is used, the compiler no longer suppresses debugging information for (compiler generated) local assembler labels. This option is useful in combination with the CrossView51 ROM debugger, which does not allow a software breakpoint (3 byte JMP MONITOR) to be overlapped with a label. So, when **–gl** is used, CrossView51 can do a better job, allowing software breakpoints on safe places only. Please note that the amount of debug information in the absolute output file will be increased, which might increase the loading time of the debugger.

When you use a ROM monitor based execution environment (e.g., CrossView51 ROM), we recommend using **–grl**. When you use a skidding emulator we recommend using **–ge**.

USAGE

The **–ge** and **–gr** options are the only sub–options of **–g** that really affect the code generated for a debugging session, inserting NOP(s) for every C–line.

Examples:

To add symbolic debug information to the output files, enter:

```
cc51 -g test.c
```

To add symbolic debug information to the output files for a skidding emulator, enter:

```
cc51 -ge test.c
```

-H

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Preprocessing**. Enter a filename in the **Include this file before source** field.

–H*file*

Arguments:

The name of an include file.

Description:

Include *file* before compiling the C source. This is the same as specifying #include "*file*" at the first line of your C source.

Example:

```
cc51 -Hstdio.h test.c
```

–I

-I

Option:

From the **Project** menu, select **Directories...**
Add one or more directory paths to the **Include Files Path** field.

–I*directory*

Arguments:

The directory of the include file.

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 directories named in **–I** options in left–to–right order. If the include file is still not found, the compiler searches in a directory specified with the environment variable CC51INC. CC51INC may contain more than one directory. Finally, the directory `../include` relative to the directory where the compiler binary is located is searched. This is the standard include directory supplied with the compiler package.

For #include files whose names are in <>, the directory of the file containing the #include line is not searched. However, the directories named in **–I** options (and the one in CC51INC and the relative path) are still searched.

Example:

```
cc51 -I/proj/include test.c
```

Section 4.3, *Include Files*

-ivo

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Code Generation**. Enable the option **Generate code for interrupt vector** and enter a vector offset value in the **Reset/interrupt vector offset (0–0xFFFF)** field.

–ivo=*value*

Arguments:

A 16–bit base address for the interrupt vector table.

Description:

This option specifies a 16–bit base address for the interrupt vector table. This option is useful when running the application on an evaluation board using the RISM ROM monitor. RISM claims non–used interrupts to user RAM.

Example:

To specify `0x4000` as the base address of the interrupt vector table, enter:

```
cc51 –ivo=0x4000 test.c
```

–v

-l / -li

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Miscellaneous**. Add the option to the **Additional C Compiler options** field.

–l[**i**]

Pragma:

#pragma listinc

Description:

Generate a list file with the name of the module and `.lst` suffix. The list file is only generated for use with debuggers using a list file instead of the C source file (e.g. an ICE5100 debugger). It is not meant as a listing generator, because other tools (e.g. pr) are available for this purpose.

Note that if the **–l** option is used all line number references (used in error messages, source merging (**–s**), __LINE__, __FILE__, object line records, etc.) are now referring to the list file.

If **–li** is used, include files are expanded in the list file. This is only useful with include files containing executable statements.

Example:

To generate the list file `test.lst`, in which include files are expanded, enter:

```
cc51 -li test.c
```

Pragmas `listinc` and `nolistinc` in the section 4.4, *Pragmas*.

-M

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Memory Model**. Choose a **Data model**.

–M*model*

Arguments:

The memory model to be used, where *model* is one of:

s small (static in data)
a auxpage (static in pdat)
l large (static in xdat)
r reentrant (in xdat)

Default:

–Ms

Description:

Select memory model to be used. The default memory model is small.

Example:

```
cc51 -Ml test.c
```

Section 3.2.2, *Memory Models*

-m

Option:

From the **Project** menu, select **Project Options...**
Expand the **Processor** entry and select **Memory**. Select a size in the **On–chip data RAM size (0–256)** field.

–m*mem=size*

Arguments:

A memory space with a memory size. *mem* can be one of:

mem	Description	Default size (bytes)
bi	_bit	128 (bits)
da	_data	128
id	_idat	128
pd	_pdat	256
xd	_xdat	65536
co	constant (_rom)	65536
ba	bitaddressable (_bdat)	16
pr	program (_rom)	65536

Table 4–3: Memory spaces

Description:

Specify the memory size (limits) to be used by the compiler for checking static memory allocations of the file being processed. The limits used and the space allocated by the module are reported when **cc51** completes compilation (unless **–t** option is used).

-misrac

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **MISRA C**.
Select a MISRA C configuration. Optionally, in the **MISRA C Rules** entry, specify the individual rules.

–misrac*n,n,....*

Arguments:

The MISRA C rules to be checked.

Description:

With this option, the MISRA C rules to be checked can be specified. Refer to Appendix A, *MISRA C*, for a list of supported and unsupported MISRA C rules.

Example:

```
cc51 –misrac9 test.c
```

Will generate an error in case 'test.c' contains nested comments.

-misrac-advisory-warnings / -misrac-required-warnings

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **MISRA C**.
Select **Generate warnings instead of errors for required rules** and/or **Generate warnings instead of errors for advisory rules**.

–misrac–advisory–warnings
–misrac–required–warnings

Description:

With this option, you can change the error level for messages on the required and advisory MISRA C rules to warnings. The default messages are errors. Refer to Appendix A, *MISRA C* for a list of MISRA C rules.

Example:

```
cc51 –misrac9 –misrac–required–warnings test.c
```

Will generate a warning in case 'test.c' contains nested comments.

-n

Option:

-n

Description:

Do not create output files; instead, the output is sent to stdout.

Example:

```
cc51 -n test.c
```

-nofastparm

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation**. Disable the option **Use fast internal RAM area for parameters**. This option is only available if you selected the auxiliary or large memory model.

–nofastparm

Description:

Switch off the use of the fast parameter area. This will shorten the interrupt frame and therefor speedup the execution speed of interrupts.

Example:

```
cc51 –nofastparm test.c
```

You can use this feature for individual interrupt functions with **#pragma intsave NOPARM***regbank*. Use this pragma only if you know that the fast parameter area is not used within the function.

Section 7.7, *Interrupt Functions*
Pragma `intsave` in the section 4.4, *Pragmas*.

-noregaddr

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Code Generation**. Disable the option **Allow absolute register addresses (AR0–AR7) in generated code**.

–noregaddr

Description:

Get register bank independent code generation. With this option it is possible to generate functions that can be called from any function independent of its register bank.

Example:

```
cc51 –noregaddr test.c
```

Section 3.17, *Register Bank Independent Code Generation*

-O

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select an **Optimization level**.

If you select **Custom optimization** in the **Optimization level** box, you can enable or disable individual optimizations in the **Custom optimizations** list.

–O*flags*

Pragma:

#pragma optimize *flags*

Arguments:

One or more optimization flags.

Default:

–O1

Description:

Control optimization. If you do not use this option, the default optimization of **cc51** is **–O1**, which is a compromise of code size and compilation speed.

Flags which are controlled by a letter, can be switched on with the lower case letter and switched off with the uppercase letter. These options are described together.

All optimization flags can also be given in the source file after a `#pragma optimize`. For example, specifying **–Oc** on the command line, is the same as specifying `#pragma optimize c` in the source file.

An overview of the flags is given below.

a – relax alias checking
c – common subexpression elimination
d – data flow, constant/copy propagation
f – optimize for speed (increases code size)
h – peephole optimization
i – move invariant code outside loop (needs **–Oc**)
k – optimize into compound assignments
l – fast loops (increases code size)
m – allow code movement
p – control flow optimization
r – use register parameter passing
s – optimize initialization loops
t – turn tentative into defining occurrence
v – loop variable optimization
w – allow register variables
0 – same as **–OACDFHIKLMNPRSTVW** (no optim)
1 – same as **–OAcdFhikLmnprstVw** (default)
2 – same as **–OacdFhikLmnprstvw** (size)
3 – same as **–Oacdfhiklmnprstvw** (speed)

Example:

```
cc51 -OAcdFhikLmprstVw test.c
```

Pragma `optimize` in the section 4.4, *Pragmas*.

-Onumber

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select an **Optimization level**.

–O*number*

Arguments:

A number in the range 0 – 3.

Default:

–O1

Description:

Control optimization. You can specify a single number in the range 0 – 3, to enable or disable optimization. The options are a combination of the other optimization flags:

–O0 – same as **–OACDFHIKLMNPRSTVW** (no optimization)
–O1 – same as **–OAcdFhikLmnprstVw** (default)
–O2 – same as **–OacdFhikLmnprstvw** (size)
–O3 – same as **–Oacdfhiklmnprstvw** (speed)

The flags 0 to 3 cannot be concatenated with other flags. For example, **–Oa2c** is not allowd, **–OacF** is allowed.

Example:

To optimize for code size, enter:

```
cc51 -O2 test.c
```

-Oa / -OA

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Relax alias checking**.

–Oa / –OA

Pragma:

#pragma noalias
#pragma alias

#pragma optimize a
#pragma optimize A

Default:

–OA

Description:

With **–Oa** you relax alias checking. If you specify this option, **cc51** will not erase remembered register contents of user variables if a write operation is done via an indirect (calculated) address. You must be sure this is not done in your C–code (check pointers!) before turning on this option.

With **–OA** you specify strict alias checking. If you specify this option, **cc51** erases all register contents of user variables when a write operation is done via an indirect (calculated) address.

Example:

An example is given in section 4.5 *Alias* in this chapter.

Pragmas `noalias`, `alias` and `optimize` in the section 4.4, *Pragmas*.

-Oc / -OC

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Common subexpression elimination (CSE)**.

–Oc / –OC

Pragma:

#pragma optimize c
#pragma optimize C

Default:

–Oc

Description:

With **–Oc** you enable CSE (common subexpression elimination). With this option specified, the compiler tries to detect common subexpressions within the C code. The common expressions are evaluated only once, and their result is temporarily held in registers or in `data`. The size of the maximum used `data` area can be specified with the **–c***size* option (default 0).

The **–Oc** option must be on to enable moving invariant code outside a loop (**–Oi**).

With **–OC** you disable CSE (common subexpression elimination). With this option specified, the compiler will not try to search for common expressions. Also moving invariant code outside a loop will be disabled.

Example:

```
/*
 * Compile with -OC -O0,
 * Compile with -Oc -O0, common subexpressions are found
 *    and temporarily saved.
 */

char x, y, a, b, c, d;

void
main( void )
{
      x = (a * b) - (c * d);

      y = (a * b) + (c * d); /*(a*b) and (c*d) are common */
}
```

–c

Pragma `optimize` in section 4.4, *Pragmas*.

-Od / -OD

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Constant and copy propagation (data flow)**.

–Od / –OD

Pragma:

#pragma optimize d
#pragma optimize D

Default:

–Od

Description:

With **–Od** you enable constant and copy propagation. With this option, the compiler tries to find assignments of constant values to a variable, a subsequent assignment of the variable to another variable can be replaced by the constant value.

With **–OD** you disable constant and copy propagation.

Example:

```
/*
 * Compile with -OD -O0, 'i' is actually assigned to 'j'
 * Compile with -Od -O0, 15 is assigned to 'j', 'i' was
 * propagated
 */

int i;
int j;

void
main( void )
{
      i = 10;
      j = i + 5;
}
```

Pragma `optimize` in section 4.4, *Pragmas*.

-Of / -OF

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Produce small code (favor code size above execution speed)**.

–Of / –OF

Pragma:

#pragma optimize f
#pragma optimize F

Pragma:

speed / size

Default:

–OF

Description:

With **–Of** you produce fast code. Favour execution speed above code density. Note that this option may increase code size. If **–Of** is specified, **cc51** uses inline code for a number of integer operations and index calculations of external RAM objects instead of emitting code which calls a run time library routine. This option is recommended for those modules which contain code which must be executed fast (e.g. interrupt functions).

With **–OF** you produce small code. Favour code density above execution speed. If **–OF** is specified, **cc51** calls a run time library routine for a number of integer operations and index calculations of external RAM objects.

Example:

```
/*
 * Compile with -OF -O0, produce small code, call
 *                       run time library routine
 * Compile with -Of -O0, produce fast code
 */

extern _xdat char arr[];

void
main( void )
{
      unsigned char c;

      arr[c] = 'x';
}
```

Pragmas **speed**, **size** and **optimize** in the section 4.4, *Pragmas*.

-Oh / -OH

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Peephole optimizer (remove redundant code)**.

–Oh / –OH

Pragma:

#pragma optimize h
#pragma optimize H

Default:

–Oh

Description:

With **–Oh** you enable peephole optimization. Remove redundant code.

With **–OH** you disable peephole optimization.

Example:

```
/*
 * Compile with -OH, unnecessary instructions found
 * Compile with -Oh, unnecessary instructions removed by
 *  peephole optimizer
 *
 * Peephole optimizer searches for patterns in the generated
 * code. E.g.
 *    MOVX @DPTR,A
 *    MOV A, ACC
 *   does not need the MOV, while the value is still in A.
 */

_xdat int i;
void
main( void )
{
      i = 0x202;
}
```

Pragma `optimize` in section 4.4, *Pragmas*.

-Oi / -OI

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Move invariant code outside loop**.

–Oi / –OI

Pragma:

#pragma optimize i
#pragma optimize I

Default:

–Oi

Description:

With **–Oi** you move invariant code outside a loop. Note that the option **–Oc** must be on to use this option.

With **–OI** you disable moving invariant code outside a loop.

Example:

```
/*
 * Compile with -OI -Oc -O0, normal cse is done
 * Compile with -Oi -Oc -O0, invariant code is found in
 *    the loop, code is moved outside the loop.
 */
void
main( void )
{
      char x, y, a, b;
      int i;

      for( i=0; i<20; i++ )
      {
            x = a + b;
            y = a + b;
      }
}
```

–Oc
Pragma `optimize` in section 4.4, *Pragmas*.

-Ok / -OK

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Optimize into compound assignments**.

–Ok / –OK

Pragma:

#pragma optimize k
#pragma optimize K

Default:

–Ok

Description:

With **–Ok** you optimize into compound assignments.

With **–OK** you do not optimize into compound assignments.

Example:

```
/*
 * Compile with -OK -O0, only second statement is compound
 *                  assignment
 * Compile with -Ok -O0, first statement is optimized in
 *                  compound assignment
 */

unsigned _pdat int i;

void
main( void )
{
      i = i + 1;
      i += 1;             /* compound assignment */
}
```

Pragma `optimize` in section 4.4, *Pragmas*.

-Ol / -OL

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Fast loops (more code size)**.

–Ol / –OL

Pragma:

#pragma optimize l
#pragma optimize L

Default:

–OL

Description:

With **–Ol** you enable fast loops. Duplicate the loop condition. Evaluate the loop condition one time outside the loop, just before entering the loop, and at the bottom of the loop. This saves one unconditional jump and gives less code inside a loop.

With **–OL** you disable fast loops.

Example:

```
/*
 * Compile with -OL -O0
 * Compile with -Ol -O0, compiler duplicates the loop
 * condition, the unconditional jump is removed.
 */
int i;

void
main( void )
{
      for( ; i<10; i++ )
      {
            do_something();
      }
}
```

Pragma `optimize` in section 4.4, *Pragmas*.

-Om / -OM

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Code order rearranging in flow optimization**.

–Om / –OM

Pragma:

#pragma optimize m
#pragma optimize M

Default:

–Om

Description:

With **–Om** you enable code rearranging. Try to move (sub)expressions to get faster code. Some debuggers may have difficulties with such options.

With **–OM** you disable code rearranging.

Example:

```
/*
 * Compile with -OM -O0, code as written sequential
 * Compile with -Om -O0, code is rearranged
 *
 * Code rearranging enables other optimizations to
 * optimize better, e.g. CSE
 */

int   i;
extern      void  dummy( void );

void main ()
{
      do
      {
            if    ( i )
            {
                  i--;
            }
            else
            {
```

```
                    i++;
                    break;
                }
                dummy();
        } while ( i );
}
```

Pragma **optimize** in section 4.4, *Pragmas*.

-Op / -OP

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Extra flow optimization pass**.

–Op / –OP

Pragma:

#pragma optimize p
#pragma optimize P

Default:

–Op

Description:

With **–Op** you enable control flow optimizations on the intermediate code representation, such as jump chaining and conditional jump reversal.

With **–OP** you disable control flow optimizations.

Example:

```
/*
 * Compile with -OP -O0
 * Compile with -Op -O0, compiler finds first time 'i' is
 * always < 10, the unconditional jump is removed.
 */
int i;

void
main( void )
{
      for( i=0; i<10; i++ )
      {
            do_something();
      }
}
```

Pragma `optimize` in section 4.4, *Pragmas*.

USAGE

-Or / -OR

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Use register parameter passing (R2–R7)**.

–Or / –OR

Pragma:

#pragma optimize r
#pragma optimize R

Default:

–Or

Description:

With **–Or** you specify to use real register parameter passing. **cc51** will treat all function prototypes and function declarations which are not explicitly programmed `_regparm` or `_cdecl` as `_regparm` functions.

With **–OR cc51** will treat all function prototypes and function declarations which are not explicitly programmed `_regparm` or `_cdecl` as `_cdecl` functions.

Example:

```
/* Compile with -OR -O0, parameters are passed in fixed memory areas
 * Compile with -Or -O0, compiler uses register parameter passing
 */
extern int func( char, int );

void
main( void )
{
      char c;
      int x, y;

      x = func( c, y );
}
```

Section *Function Parameters* for details on parameter passing.
Pragma `optimize` in section 4.4, *Pragmas*.

-Os / -OS

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Optimize initialization loops**.

–Os / **–OS**

Pragma:

#pragma optimize s
#pragma optimize S

Default:

–Os

Description:

With **–Os** you optimize initialization loops. Replace an initialization loop by a `_memset()` routine.

With **–OS** you do not optimize initialization loops.

Example:

```
/*
 * Compile with -OS, loop code is found
 * Compile with -Os, loop is found to be an initialization
 * loop, code is generated as for the inline '_memset' function
 */

char arr[20];

void
main( void )
{
      unsigned char c;

      for( c=0; c<20; c++ )
            arr[c] = 0;

      _memset( arr, 0, 20 );
}
```

Pragma `optimize` in section 4.4, *Pragmas*.

-Ot / -OT

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Keep variables in the order of declaration**.

–Ot / –OT

Pragma:

#pragma optimize t
#pragma optimize T

Default:

–Ot

Description:

With **–Ot** you allocate a tentative object at the point of programming. Normally all tentative object allocations are delayed by the compiler until the end of the module. With this option set, the allocation is done immediately. This enables the compiler to optimize access to the objects. Objects are forced to remain in the same order.

With **–OT** you keep a tentative declaration tentative until the end of the module. The compiler cannot optimize access to these objects.

Example:

```
/*
 * Compile with -OT -O0, space for 'a' and 'b' is
 *    allocated at end of module
 * Compile with -Od -O0, space for 'a' and 'b' is
 *    allocated at point of programming. compiler knows
 *    that 'b' is located after 'a', so DPTR can be
 *    incremented
 */
_xdat char a;
_xdat char b;

void
main( void )
{
      char c;

      c = a + 5;
      b = 10;
}
```

Pragma **optimize** in section 4.4, *Pragmas*.

-Ov / -OV

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Loop variable optimization**.

–Ov / –OV

Pragma:

#pragma optimize v
#pragma optimize V

Default:

–OV

Description:

With **–Ov** you enable loop variable detection. With this option specified, the compiler tries to detect within a loop which variable is the loop variable. When possible, this variable is temporarily moved into registers.

With **–OV** you disable loop variable detection.

Example:

```
/*
 * Compile with -OV -O0, loop variable 'i' is an automatic
 * Compile with -Ov -O0, loop variable 'i' is temporarily
 *                       moved into registers.
 */
int j;

void
main( void )
{
      int i;
      for( i=0; i<10; i++ )
      {
            j = 2 * i;
      }
}
```

–Oc

Pragma `optimize` in section 4.4, *Pragmas*.

-Ow / -OW

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select the **Custom optimization** level. Enable or disable the option **Allow register automatic variables (R0–R7)**.

–Ow / –OW

Pragma:

#pragma optimize w
#pragma optimize W

Default:

–Ow

Description:

With **–Ow** you allow **cc51** to use register variables. Of course this cannot be done when all registers are in use.

With **–OW** you do not allow **cc51** to use register variables.

Example:

```
/*
 * Compile with -OW -O0, variables are not allowed in registers
 * Compile with -Ow -O0, variables are allowed in registers.
 *  'k' is still an automatic, because all registers are in use
 */

void
main( void )
{
      int i, j, k;

      k = i * j;
}
```

Pragma `optimize` in section 4.4, *Pragmas*.

USAGE

-o

Option:

EDE determines the name of the output file with the same basename as the source file and extension `.src`.

–o *file*

Arguments:

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

Default:

Module name with `.src` suffix.

Description:

Use name as output file *name*, instead of the module name with `.src` suffix. Special care must be taken when using this option, the first **–o** option found acts on the first file to compile, the second **–o** option acts on the second file to compile, etc.

Example:

When specified:

```
cc51 file1.c file2.c –o file3.src –o file2.src
```

two files will be created, **file3.src** for the compiled file **file1.c** and **file2.src** for the compiled file **file2.c**.

-pa / -pd / -pp / -ps

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Code Generation**. Select one of the **Data Pointer** options.

–p{a|d|p|s}

Description:

The standard 8051 architecture provides just one 16–bit pointer for indirect addressing of external memory (DPTR). At this moment there are several architectures supporting more than just one data pointer. The Infineon Technologies C500/C800 family has support for 8 16–bit data pointers (**–ps**). The following derivatives have support for 2 16–bit data pointers: the Atmel AT8x53/AT89S53/AT89S4D12/AT89S8252 (**–pa**), the Dallas 80C320/520/530 and AMD 80C521 (**–pd**), and the Philips 51 family (**–pp**).

Example:

To specify to use 8 16–bit data pointers, enter:

```
cc51 -ps test.c
```

Section *Multiple Data Pointer Support* in chapter *Run–time Environment*.

-R

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry, expand the **Code Generation** entry and select **Segment names**. Select a memory space and optionally enter a new name.

–R*mem*[**=***name*]

Arguments:

A memory space, optionally followed by a segment name. *mem* can be one of:

mem	**Description**
bi	_bit
da	_data
id	_idat
pd	_pdat
xd	_xdat
co	constant (_rom)
ba	bitaddressable (_bdat)
pr	program (_rom). Note that the segment name can only be changed in the reentrant memory model.

Table 4–4: Memory spaces

Description:

The compiler defaults to a segment naming convention, using '**C51_**' and a two letter memory type abbreviation: C51_PR for executable code (only in reentrant model), C51_XD for static external RAM etc. When you select code bank switching (option **–banks**) the compiler automatically uses a different segment name convention for executable code using the module name to allow for proper function distribution over different code banks.

In static models the default segment naming for executable code uses the module name instead of 'C51'.

In case a module must be loaded at a fixed address or a data segment needs a special place in memory, the **–R** option enables you to generate a unique segment name. In this way the order LINK51 allocates these segments can be specified in a linker command file.

If *mem* is specified without a name, the module name is used instead of C51_, otherwise *name*_ is used.

Notice, that when rom size medium is used (option **–rm**), the compiler implies a **–Rpr**, which means a (unique) code segment name is emitted using the module name and having the INBLOCK attribute.

mem is the same as specified with the –**m** option.

Example:

To create a new segment name (`NEW_DA`) for DATA segments, enter:

```
cc51 –Rda=NEW test.c
```

–banks, –r

-r

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Memory Model**. Choose a **Code size limits (ROM model)** option.

–**r***romsize*

Arguments:

A single character specifying the rom size. *romsize* can be one of:

s (small: 2K program)
m (medium: 2K modules, 64K program)
l (large: 64K modules, 64K program)

Default:

–rl

Description:

Select the way function calls and non short jumps are generated. **cc51** always tries to use the `sjmp` instruction. If the distance is larger than 127 bytes, either an `ajmp`, generic `jmp` or `ljmp` is emitted. For function calls (including library calls) the same decision is made: `acall`, generic `call` or `lcall`. Code generated for small model is more efficient than medium model, which in turn is more efficient than large model:

s The complete application (C run time library included) is assumed to fit into one 2K byte block. The compiler emits `acall` for `calls` and `ajmp` for non short jumps.

m This option implies a **–Rpr**, which means **cc51** uses the module name for the declaration of the CODE segment and assigns this segment the INBLOCK attribute. This means **link51** allocates this segment within a 2K bank. The size of the C module may of course not exceed 2048 bytes. **cc51** emits generic `call` and `jmp` instructions, which are translated by **asm51** into `acall`/`ajmp` for calls and jumps within the module (read within the same segment), and to `lcall`/`ljmp` for external calls and jumps. This approach is only useful with a large number of 'small' (approx. 400 bytes) C modules, because **link51** must allocate each of these segments within a 2K bank. If larger modules are

used, a gap may be introduced. When using rom size medium, **cc51** reports code size with a 'high estimate' (worst case) remark, because it does not know how the generic `call`/`jmp` instructions are going to be translated by the assembler.

l The C module will be larger than 2K bytes. The compiler emits `lcall` and `ljmp` instructions.

The **-rm** option is ignored for all static models.

Example:

```
cc51 -rm -Mr test.c
```

-S

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation**. Select the **Keep strings in ROM (use _rom keyword for a pointer to a string)** radio button.

–S

Pragma:

#pragma romstring

Description:

Put strings in ROM only.

By default, strings are allocated in both ROM and RAM. Strings are copied from ROM to RAM by the C startup code.

Example:

```
cc51 -S test.c
```

Pragmas `romstring` and `ramstring` in the section 4.4, *Pragmas*.
Section *Strings*

-s

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Miscellaneous**. Enable the option **Merge C source code with assembly in output file (.src)**.

-s

Pragma:

#pragma source

Description:

Merge C source code with generated assembly code in output file.

Example:

```
cc51 -s test.c

; test.c       3 int    i;
         PUBLIC  _i
_i:      DS      2
; test.c       4
; test.c       5 main ()
; test.c       6 {
         PUBLIC  _?main
TEST_MAIN_PR     SEGMENT CODE
         RSEG    TEST_MAIN_PR
_?main:
```

Pragmas `source` and `nosource` in the section 4.4, *Pragmas*.

-se

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Language Extensions**.
Enable the option **Allow enum objects of type 'char'**.

–se

Description:

Treat small enumerated types as `char` instead of `int`.

Example:

```
cc51 -se test.c
```

-shiftright-signfill

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Code Generation**.
Enable the option **Use sign fill on signed shift right**.

–shiftright–signfill

Description:

With a right–shift operation on a signed value, the compiler by default fills the vacant bits with zero. With this option the compiler uses the value of the sign bit instead.

Example:

```
/* by default the right-shift results in 511
 * with -shiftright-signfill the result is -1
 */
void f( void )
{
    int x = -128;
    x >>= 7;
}
```

-t

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Miscellaneous**. Enable the option **Display module summary**.

-t

Description:

Produce totals (module summary).

Example:

```
cc51 -t test.c

8051 C compiler vx.y rz      SN00000000-015 (c) year TASKING, Inc.

MODULE SUMMARY          STATIC      OVERLAYABLE (LIMIT)

Code size   =     49          (65536)
Constant size     =     0           (65536)
Direct variable size    =     4     0     (  128)
Indirect variable size  =     0     0     (  128)
Paged auxiliary size    =     0     0     (  256)
Bit size    =     1     0     (  128)
Bit addressable size    =     2     0     (   16)
Auxiliary variable size =     0     0     (65536)
Interrupts used:
Register banks used:          0

processed 25 lines at 7075 lines/min
```

-U

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Preprocessing**. Undefine one or more of the predefined symbols _MODEL or _CC51 by disabling the corresponding option.

–U*name*

Arguments:

The name macro you want to undefine.

Description:

Remove any initial definition of identifier *name* as in #undef, unless it is a predefined ANSI standard macro. ANSI specifies the following predefined symbols to exist, which cannot be removed:

__FILE__	”current source filename”
__LINE__	current source line number (int type)
__TIME__	”hh:mm:ss”
__DATE__	”Mmm dd yyyy”
__STDC__	level of ANSI standard (because **cc51** has a number of language extensions and 8051 specific implementations, this value is set to 0)

When **cc51** is invoked, also the following predefined symbols exist:

_CC51	value represents version of TASKING C 8051 compiler.
_MODEL	memory model used (see section 3.2.2, *Memory Models* for details)

These symbols can be turned off with the **–U** option.

Example:

```
cc51 –UNORAM test.c
```

–D

-u

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Language Extensions**.
Enable the option **Treat 'char' variables as unsigned**.

–u

Description:

Treat 'character' type variables as 'unsigned character' variables. By default `char` is the same as specifying `signed char`. With **–u** `char` is the same as `unsigned char`.

Example:

With the following command `char` is treated as `unsigned char`:

```
cc51 –u test.c
```

-V

Option:

–V

Description:

Display version information.

Example:

```
cc51 -V

TASKING 8051 C compiler     vx.yrz Build nnn
Copyright years Altium BV   Serial# 00000000
```

-v

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Code Generation**. Disable the option **Generate code for interrupt vector**.

–v

Pragma:

#pragma novector

Description:

Do not generate code for interrupt vector and reference to interrupt handler in the run–time library.

This option also disables option **–ivo**.

Example:

```
cc51 -v test.c
```

Options **–ivo** and **–vf**

Pragmas `vector` and `novector` in the section 4.4, *Pragmas*.

Section 7.7, *Interrupt Functions* in chapter *Run–time Environment*.

-vf

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Code Generation**. Disable the option **Generate frame for interrupt handler**.

–vf

Description:

Do not generate interrupt frame for interrupt handler.

Example:

```
cc51 -vf test.c
```

Section 7.7, *Interrupt Functions* in chapter *Run–time Environment*.

-vo

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Miscellaneous**. Add the option to the **Additional C Compiler options** field.

-vo

Description:

Generate old style interrupt frame. This option is available for backwards compatibility only.

Example:

```
cc51 -vo test.c
```

Section 7.7, *Interrupt Functions* in chapter *Run–time Environment*.

-w / -wstrict

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Warnings**.
Enable one of the options **Report all warnings**, **Suppress all warnings**, or **Suppress specific warnings** and enter the numbers, separated by commas, of the warnings you want to suppress. Optionally enable the option **Issue strict warnings**.

–w[*num*]
–wstrict

Arguments:

Optionally the warning number to suppress.

Description:

–w suppress all warning messages. –**w***num* only suppresses the given warning. **–wstrict** suppresses all ”strict” warning messages (183, 196).

Example:

To suppress warning 135, enter:

```
cc51 file1.c –w135
```

-x

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation**. Enter a size in the **Amount of DATA for automatics (overlayable)** field.

–x*size*

Pragma:

#pragma extend *size*

Arguments:

A number of bytes.

Default:

–x4

Description:

Extend amount of internal RAM for automatics.

Example:

To reserve 10 bytes of internal RAM for automatics, enter:

```
cc51 –x10 test.c
```

Pragma `extend` in the section 4.4, *Pragmas*.

Option **–c**

-z

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Miscellaneous**. Add the option to the **Additional C Compiler options** field.

-z*pragma*

Arguments:

A pragma as listed in section 4.4, *Pragmas*.

Description:

With this option you can give a pragma on the command line. This is the same as specifying '#pragma *pragma*' in the C source. Dashes ('–') on the command line in the pragma are converted to spaces. A dash prefixed by another dash or space is never translated, so it is still possible to specify a dash for negative numbers as pragma argument.

Example:

To issue a '#pragma intsave R0' using the command line, enter:

```
cc51 -zintsave-R0 file.c
```

The '–' between `insave` and `R0` is converted to a space.

Section 4.4, *Pragmas*.

4.3 INCLUDE FILES

You may specify include files in two ways: enclosed in <> or enclosed in "". When an #include directive is seen, **cc51** uses the following algorithm trying to open the include file:

1. If the filename is enclosed in "", and it is not an absolute pathname (does not begin with a '\' for PC, or a '/' for UNIX), the include file is searched for in the directory of the file containing the #include line. For example, in:

   PC:

   ```
   cc51 ..\..\source\test.c
   ```

   UNIX:

   ```
   cc51 ../../source/test.c
   ```

   cc51 first searches in the directory `..\..\source` (`../../source` for UNIX) for include files.

   If you compile a source file in the directory where the file is located (**cc51 test.c**), the compiler searches for include files in the current directory.

   This first step is not done for include files enclosed in <>.

2. Use the directories specified with the **–I** options, in a left–to–right order. For example:

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

   PC:

   ```
   cc51 –I..\..\include message.c
   ```

   UNIX:

   ```
   cc51 –I../../include message.c
   ```

3. Check if the environment variable CC51INC exists. If it does, use the contents as a directory specifier for include files. You can specify more than one directory in the environment variable CC51INC by using a separator character. Instead of using **–I** as in the example above, you can specify the same directory using CC51INC:

PC:

```
set CC51INC=..\..\include
cc51 message.c
```

UNIX:

if using the Bourne shell (sh)

```
CC51INC=../../include
export CC51INC
cc51 message.c
```

or if using the C–shell (csh)

```
setenv CC51INC ../../include
cc51 message.c
```

4. When an include file is not found with the rules mentioned above, the compiler tries the subdirectory `include`, one directory higher than the directory containing the **cc51** binary. For example:

   PC:

   cc51.exe is installed in the directory `C:\CC51\BIN`
   The directory searched for the include file is `C:\CC51\INCLUDE`

   UNIX:

   cc51 is installed in the directory `/usr/local/cc51/bin`
   The directory searched for the include file is
   `/usr/local/cc51/include`

   The compiler determines run–time which directory the binary is executed from to find this `include` directory.

   A directory name specified with the **–I** option or in CC51INC may or may not be terminated with a directory separator, because **cc51** inserts this separator, if omitted.

   When you specify more than one directory to the environment variable CC51INC, you have to use one of the following separator characters:

   PC:

   `;` `,` *space*

   e.g. **`set CC51INC=..\..\include;\proj\include`**

USAGE

UNIX:

: ; , *space*

e.g. **setenv CC51INC ../../include:/project/include**

4.4 PRAGMAS

According to ANSI (3.8.6) a preprocessing directive of the form:

```
#pragma pragma-token-list new-line
```

causes the compiler to behave in an implementation–defined manner. The compiler ignores pragmas which are not mentioned in the list below. Pragmas give directions to the code generator of the compiler. Besides the pragmas there are two other possibilities to steer the code generator: command line options and keywords. The compiler acknowledges these three groups using the following rule:

Command line options can be overruled by keywords and pragmas. Keywords can be overruled by pragmas. So, pragmas have the highest priority.

This approach makes it possible to set a default optimization level for a source module, which can be overridden temporarily within the source by a pragma. For example, on the command line you have given option **–O1** (default optimizations). You can, for example, disable CSE checking in (part of) a source module by specifying `#pragma optimize C` in the source.

cc51 supports the following pragmas:

alias

Default. Same as **–OA** option. Perform strict alias checking. See also section 4.5 *Alias*.

noalias

Same as **–Oa** option. Relax alias checking.

***arglist** size*

Same as **–a** option. Change the allocation size for variable argument lists. Only useful in the static models.

asm

Insert the following (non preprocessor lines) as assembly language source code into the output file. The inserted lines are not checked for their syntax.

asm_noflush

Same as **asm**, except that the peephole optimizer does not flush the code buffer and assumes register contents remain valid.

endasm

Switch back to the C language.

cse *size*

Same as **–c** option. Change the maximum allocation size to store CSE values in.

extend *size*

Same as **–x** option. Specify the maximum size of internal RAM to be used for 'automatic variables' to be made 'register variables'. See also section 3.6, *Automatic Variables*.

intsave *registers*

When using assembly in an interrupt function you can save registers that are not automatically saved by the compiler. See also section 7.7 *Interrupt Functions* in chapter *Run–time Environment*.

listinc

Same as **–li** option. Expand include files in generated list file. Only useful with **–l** option and include files containing executable statements.

nolistinc

Default. Do not expand include files in list file.

linear_switch

Force the compiler to generate linear jump code for switch statements. See also section 3.21, *Switch Statement*.

jump_switch

Force the compiler to generate jump tables for switch statements. See also section 3.21, *Switch Statement*.

binary_switch

Force the compiler to generate binary search tables for switch statements. See also section 3.21, *Switch Statement*.

smart_switch

Default. The compiler decides what code to generate on a switch statement. In general, the compiler chooses the smallest method. See also section 3.21, *Switch Statement*.

***message** "string" ...*
message ("string" ...)

Print the message string(s) on standard output during the build process. For example:

```
#pragma message( "Compiling file " __FILE__ )
#ifdef SHOW_DATE_AND_TIME
# pragma message( " date: " __DATE__ ", time: " __TIME__ )
#endif
```

***optimize** flags*

Controls the amount of optimization. The remainder of the source line is scanned for option characters, which are processed like the flags of the **–O** command line option. Please refer to the **–O** option for a list of available flags.

For example, specifying **–Oc** on the command line, is the same as specifying `#pragma optimize c` at the beginning of a source file.

page
nopage

Align or do not align code segments on a 256 byte page boundary.

In the following example, the rom variable `rdata` is placed in a segment with the PAGE attribute.

```
#pragma page
_rom char rdata[] = {1,2,3,4};
#pragma nopage
```

Note that in some situations the segment definition may be postponed till later in the code generation process. So, it could be generated after a following `#pragma nopage`. In such situations the `#pragma optimize t` (tentative declaration) may be required.

ramstring

Default. Allocate strings in ROM and RAM. The strings are copied to RAM at startup.

romstring

Same as **–S** option. Allocate strings in ROM only.

size

Default. Same as **–OF** option. Favour code density above execution speed.

speed

Same as **–Of** option. Favour execution speed above code density.

source

Same as **–s** option. Enable mixing C source with assembly code.

nosource

Default. Disable generation of C source within assembly code.

vector *[value]*

Default. Restore generation of interrupt vector after a **novector** pragma. With the *value* this pragma is the same as the **–ivo** option

novector

Same as **–v** option. Do not emit interrupt vector and reference to interrupt handler in run–time library.

4.5 ALIAS

By default the compiler assumes that each pointer may point to any object created in the program, so when any pointer is dereferenced, all register contents are assumed to be invalid afterwards.

When it is known that aliasing problems do not occur in the written C–source, alias checking may be relaxed (use the **–Oa** option or **#pragma noalias**). Relaxing alias checking may reduce code size.

Example 1:

```
void
func( int i )
{
   char     * p;
   char     c;
   char     d;

   if( i )
      p = &c;
   else
      p = &d;

   c = 2;
   d = 3;

   *p = 4;  /* may write to 'c' or 'd'          */
            /* --> aliasing object 'c' or 'd'   */

   i = c;   /* '*p' might have changed the value of 'c',    */
            /* so 'c' may not be used from register         */
            /* contents, but MUST be read from memory       */
            /* --> alias checking MUST be ON in this case   */
}
```

Example 2:

```
void
func( int i, char *p )
{
   char     c;
   char     d;

   c = 2;
   d = 3;

   *p = 4;  /* cannot write to 'c' or 'd', but to some other object
*/

   i = c;   /* '*p' cannot have changed the value of 'c',   */
            /* so 'c' may be used from register contents    */
            /* --> alias checking may be OFF in this case   */
}
```

4.6 COMPILER LIMITS

The ANSI C standard [1–2.2.4] defines a number of translation limits, which a C compiler must support to conform to the standard. The standard states that a compiler implementation should be able to translate and execute a program that contains at least one instance of every one of the following limits, (**cc51**'s actual limits are given within parentheses):

- 15 nesting levels of compound statements, iteration control structures and selection control structures
- 8 nesting levels of conditional inclusion (50)
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration (12)
- 31 nesting levels of parenthesized declarators within a full declarator
- 32 nesting levels of parenthesized expressions within a full expression
- 31 significant characters in an external identifier (full ANSI–C mode),
  120 significant characters in an external identifier (non ANSI–C mode)
- 511 external identifiers in one translation unit
- 127 identifiers with block scope declared in one block
- 1024 macro identifiers simultaneously defined in one translation unit
- 31 parameters in one function declaration
- 31 arguments in one function call
- 31 parameters in one macro definition
- 31 arguments in one macro call
- 509 characters in a logical source line (1500)
- 509 characters in a character string literal or wide string literal (after concatenation) (1500)
- 8 nesting levels for **#include**d files (50)
- 257 case labels for a switch statement, excluding those for any nested switch statements
- 127 members in a single structure or union
- 127 enumeration constants in a single enumeration
- 15 levels of nested structure or union definitions in a single struct–declaration–list

As far as the compiler implementation uses fixed tables, they will be large enough to meet the standards limits. However, most of the internal structures and tables of the compiler are dynamic. Thus the actual compiler limits are determined by the amount of free memory in the system.

CHAPTER 5

COMPILER DIAGNOSTICS

TASKING

5 CHAPTER

5.1 INTRODUCTION

cc51 has three classes of messages: user errors, warnings and internal compiler errors.

Some user error messages carry extra information, which is displayed by the compiler after the normal message. The messages with extra information are marked with 'I' in the list below. They never appear without a previous error message and error number. The number of the information message is not important, and therefore, this number is not displayed. A user error can also be fatal (marked as 'F' in the list below), which means that the compiler aborts compilation immediately after displaying the error message and may generate a 'not complete' output file.

The error numbers and warning numbers are divided in two groups. The frontend part of the compiler uses numbers in the range 0 to 499, whereas the backend (code generator) part of the compiler uses numbers in the range 500 and higher. Note that most error messages and warning messages are produced by the frontend.

Errors can be written directly to an error list file by using the **–err** option of the compiler. See also the chapter *Compiler Use*.

If you program a non fatal error, **cc51** displays the C source line that contains the error, the error number and the error message on the screen. If the error is generated by the code generator, the C source line displayed always is the last line of the current C function, because code generation is started when the end of the function is reached by the frontend. **cc51** displays the line number causing the error before the error message. **cc51** always generates the error number in the assembly output file, exactly matching the place where the error occurred.

For example, the following program, `bug.c`, causes a code generator error message:

```
      void
      main(void)
{
           char c;

           _testclear( c );
      }

E 544: (line 6) illegal testclear argument
```

The output file, `bug.src`, contains:

```
; bug.c       6          _testclear( c );
        ERROR   C51_ERROR_544
```

So, when a compilation is not successful, the generated output file is not accepted by the assembler, thus preventing a corrupt application to be made (see also the **–e** option).

Warning messages do not result in an erroneous assembly output file. They are meant to draw your attention to assumptions of the compiler, for a not correct situation. You can control warning messages with the **–w**[*num*] option.

The last class of messages are the internal compiler errors. The following format is used:

```
S number: internal error – please report
```

These errors are caused by failed internal consistency checks and should never occur. However, if such a 'SYSTEM' error appears, please report the occurrence to TASKING, using a Problem Report form. Please include a diskette or tape, containing a small C program causing the error.

5.2 RETURN VALUES

cc51 returns an exit status to the operating system environment for testing.

For example,

in a MS–DOS BATCH–file you can examine the exit status of the program executed with ERRORLEVEL:

```
cc51 –s %1.c
IF ERRORLEVEL 1 GOTO STOP_BATCH
```

In a bourne shell script, the exit status can be found in the **$?** variable, for example:

```
cc51 $*
case $? in
0)        echo ok ;;
1|2|3)    echo error ;;
esac
```

The exit status of **cc51** is one of the numbers of the following list:

0 Compilation successful, no errors
1 There were user errors, but terminated normally
2 A fatal error, or System error occurred, premature ending
3 Stopped due to user abort

5.3 ERRORS AND WARNINGS

Errors start with an error type, followed by a number and a message. The error type is indicated by a letter:

I information
E error
F fatal error
S internal compiler error
W warning

Frontend

F 1 evaluation expired

Your product evaluation period has expired. Contact your local TASKING office for the official product.

W 2 unrecognized option: '*option*'

The option you specified does not exist. Check the invocation syntax for the correct option.

E 4 expected *number* more '#endif'

The preprocessor part of the compiler found the'#if', '#ifdef' or '#ifndef' dirctive but did not find a corresponding '#endif' in the same source file. Check your source file that each '#if', '#ifdef' or '#ifndef' has a corresponding '#endif'.

E 5 no source modules

You must specify at least one source file to compile.

F 6 cannot create "*file*"

The output file or temporary file could not be created. Check if you have sufficient disk space and if you have write permissions in the specified directory.

F 7 cannot open "*file*"

Check if the file you specified really exists. Maybe you misspelled the name, or the file is in another directory.

F 8 attempt to overwrite input file "*file*"

The output file must have a different name than the input file.

E 9 unterminated constant character or string

This error can occur when you specify a string without a closing double–quote (`"`) or when you specify a character constant without a closing single–quote (`'`). This error message is often preceded by one or more E 19 error messages.

F 11 file stack overflow

This error occurs if the maximum nesting depth (50) of file inclusion is reached. Check for #include files that contain other #include files. Try to split the nested files into simpler files.

F 12 memory allocation error

All free space has been used. Free up some memory by removing any resident programs, divid the file into several smaller source files, break expressions into smaller subexpressions or put in more memory.

W 13 prototype after forward call or old style declaration – ignored

Check that a prototype for each function is present before the actual call.

E 14 ';' inserted

An expression statement needs a semicolon. For example, after `++i` in `{ int i; ++i }`.

E 15 missing filename after –o option

The **–o** option must be followed by an output filename.

E 16 bad numerical constant

A constant must conform to its syntax. For example, `08` violates the octal digit syntax. Also, a constant may not be too large to be represented in the type to which it was assigned. For example, `int i = 0x1234567890;` is too large to fit in an integer.

E 17 string too long

This error occurs if the maximum string size (1500) is reached. Reduce the size of the string.

E 18 illegal character (0x*hexnumber*)

The character with the hexadecimal ASCII value 0x*hexnumber* is not allowed here. For example, the '#' character, with hexadecimal value 0x23, to be used as a preprocessor command, may not be preceded by non–white space characters. The following is an example of this error:

```
char *s = #S ;  // error
```

E 19 newline character in constant

The newline character can appear in a character constant or string constant only when it is preceded by a backslash (\). To break a string that is on two lines in the source file, do one of the following:

- End the first line with the line–continuation character, a backslash (\).
- Close the string on the first line with a double quotation mark, and open the string on the next line with another quotation mark.

E 20 empty character constant

A character contant must contain exactly one character. Empty character contants (`''`) are not allowed.

E 21 character constant overflow

A character contant must contain exactly one character. Note that an escape sequence (for example, **t** for tab) is converted to a single character.

E 22 '#define' without valid identifier

You have to supply an identifier after a '#define'.

E 23 '#else' without '#if'

'#else' can only be used within a corresponding '#if', '#ifdef' or '#ifndef' construct. Make sure that there is a '#if', '#ifdef' or '#ifndef' statement in effect before this statement.

E 24 '#endif' without matching '#if'

'#endif' appeared without a matching '#if', '#ifdef' or '#ifndef' preprocessor directive. Make sure that there is a matching '#endif' for each '#if', '#ifdef' and '#ifndef' statement.

E 25 missing or zero line number

'#line' requires a non–zero line number specification.

E 26 undefined control

A control line (line with a '*#identifier*') must contain one of the known preprocessor directives.

W 27 unexpected text after control

'#ifdef' and '#ifndef' require only one identifier. Also, '#else' and '#endif' only have a newline. '#undef' requires exactly one identifier.

W 28 empty program

The source file must contain at least one external definition. A source file with nothing but comments is considered an empty program.

E 29 bad '#include' syntax

A '#include' must be followed by a valid header name syntax. For example, `#include <stdio.h` misses the closing '>'.

E 30 include file "*file*" not found

Be sure you have specified an existing include file after a '#include' directive. Make sure you have specified the correct path for the file.

E 31 end–of–file encountered inside comment

The compiler found the end of a file while scanning a comment. Probably a comment was not terminated. Do not forget a closing comment '*/' when using ANSI–C style comments.

E 32 argument mismatch for macro "*name*"

The number of arguments in invocation of a function–like macro must agree with the number of parameters in the definition. Also, invocation of a function–like macro requires a terminating ")" token. The following are examples of this error:

```
#define A(a) 1
int i = A(1,2); /* error */

#define B(b) 1
int j = B(1;    /* error */
```

E 33 "*name*" redefined

The given identifier was defined more than once, or a subsequent declaration differed from a previous one. The following examples generate this error:

```
int  i;
char i;          /* error */
main()
{
}
```

```
main()
{
     int j;
     int j;     /* error */
}
```

W 34 illegal redefinition of macro "*name*"

A macro can be redefined only if the body of the redefined macro is exactly the same as the body of the originally defined macro.

This warning can be caused by defining a macro on the command line and in the source with a '#define' directive. It also can be caused by macros imported from include files. To eliminate the warning, either remove one of the definitions or use an '#undef' directive before the second definition.

E 35 bad filename in '#line'

The string literal of a #line (if present) may not be a "wide–char" string. So, `#line 9999 L"t45.c"` is not allowed.

W 36 'debug' facility not installed

'#pragma debug' is only allowed in the debug version of the compiler.

W 37 attempt to divide by zero

A divide or modulo by zero was found. Adjust the expression or test if the second operand of a divide or modulo is zero.

E 38 +non integral switch expression

A `switch` condition expression must evaluate to an integral value. So, `char *p = 0; switch (p)` is not allowed.

F 39 unknown error number: *number*

This error may not occur. If it does, contact your local TASKING office and provide them with the exact error message.

W 40 non–standard escape sequence

Check the spelling of your escape sequence (a backslash, \, followed by a number or letter), it contains an illegal escape character. For example, `\c` causes this warning.

E 41 '#elif' without '#if'

The '#elif' directive did not appear within an '#if', '#ifdef or '#ifndef' construct. Make sure that there is a corresponding '#if', '#ifdef' or '#ifndef' statement in effect before this statement.

E 42 syntax error, expecting parameter type/declaration/statement

A syntax error occurred in a parameter list a declaration or a statement. This can have many causes, such as, errors in syntax of numbers, usage of reserved words, operator errors, missing parameter types, missing tokens.

E 43 unrecoverable syntax error, skipping to end of file

The compiler found an error from which it could not recover. This error is in most cases preceded by another error. Usually, error E 42.

I 44 in initializer "*name*"

Informational message when checking for a proper constant initializer.

E 46 cannot hold that many operands

The value stack may not exceed 20 operands.

E 47 missing operator

An operator was expected in the expression.

E 48 missing right parenthesis

')' was expected.

W 49 attempt to divide by zero – potential run–time error

An expression with a divide or modulo by zero was found. Adjust the expression or test if the second operand of a divide or modulo is zero.

E 50 missing left parenthesis

'(' was expected.

E 51 cannot hold that many operators

The state stack may not exceed 20 operators.

E 52 missing operand

An operand was expected.

E 53 missing identifier after 'defined' operator

An identifier is required in a `#if defined(`*identifier*`)`.

E 54 +non scalar controlling expression

Iteration conditions and 'if' conditions must have a scalar type (not a struct, union or a pointer). For example, after `static struct {int i;} si = {0};` it is not allowed to specify `while (si) ++si.i;`.

E 55 operand has not integer type

The operand of a '#if' directive must evaluate to an integral constant. So, `#if 1.` is not allowed.

W 56 '*<debugoption><level>*' no associated action

This warning can only appear in the debug version of the compiler. There is no associated debug action with the specified debug option and level.

W 58 invalid warning number: *number*

The warning number you supplied to the **-w** option does not exist. Replace it with the correct number.

F 59 sorry, more than *number* errors

Compilation stops if there are more than 40 errors.

E 60 label "*label*" multiple defined

A label can be defined only once in the same function. The following is an example of this error:

```
f()
{
lab1:
lab1:       /* error */
}
```

E 61 type clash

The compiler found conflicting types. For example, a `long` is only allowed on `int` or `double`, no specifiers are allowed with `struct`, `union` or `enum`. The following is an example of this error:

```
unsigned signed int i;      /* error */
```

E 62 bad storage class for "*name*"

The storage class specifiers `auto` and `register` may not appear in declaration specifiers of external definitions. Also, the only storage class specifier allowed in a parameter declaration is `register`.

E 63 "*name*" redeclared

The specified identifier was already declared. The compiler uses the second declaration. The following is an example of this error:

```
struct T { int i; };
struct T { long j; };   /* error */
```

E 64 incompatible redeclaration of "*name*"

The specified identifier was already declared. All declarations in the same function or module that refer to the same object or function must specify compatible types. The following is an example of this error:

```
f()
{
     int i;
     char i;              /* error */
}
```

W 66 function "*name*": variable "*name*" not used

A variable is declared which is never used. You can remove this unused variable or you can use the **–w66** option to suppress this warning.

W 67 illegal suboption: *option*

The suboption is not valid for this option. Check the invocation syntax for a list of all available suboptions.

W 68 function "*name*": parameter "*name*" not used

A function parameter is declared which is never used. You can remove this unused parameter or you can use the **–w68** option to suppress this warning.

E 69 declaration contains more than one basic type specifier

Type specifiers may not be repeated. The following is an example of this error:

```
int char i;     /* error */
```

E 70 +'break' outside loop or switch

A `break` statement may only appear in a `switch` or a loop (`do`, `for` or `while`). So, `if (0) break;` is not allowed.

E 71 illegal type specified

The type you specified is not allowed in this context. For example, you cannot use the type **void** to declare a variable. The following is an example of this error:

```
void i;                 /* error */
```

W 72 duplicate type modifier

Type qualifiers may not be repeated in a specifier list or qualifier list. The following is an example of this warning:

```
{ long long i; }      /* error */
```

E 73 object cannot be bound to multiple memories

Use only one memory attribute per object. For example, specifying both rom and ram to the same object is not allowed.

E 74 declaration contains more than one class specifier

A declaration may contain at most one storage class specifier. So, **register auto i;** is not allowed.

E 75 +'continue' outside a loop

continue may only appear in a loop body (**do**, **for** or **while**). So, **switch (i) {default: continue;}** is not allowed.

E 76 duplicate macro parameter "*name*"

The given identifier was used more than one in the formatl parameter list of a macro definition. Each macro parameter must be uniquely declared.

E 77 parameter list should be empty

An identifier list, not part of a function definition, must be empty. For example, **int f (i, j, k);** is not allowed on declaration level.

E 78 'void' should be the only parameter

Within a function protoype of a function that does not except any arguments, **void** may be the only parameter. So, **int f(void, int);** is not allowed.

E 79 +constant expression expected

A constant expression may not contain a comma. Also, the bit field width, an expression that defines an enum, array–bound constants and **switch** case expressions must all be integral contstant expressions.

E 80 '#' operator shall be followed by macro parameter

The '#' operator must be followed by a macro argument.

E 81 '##' operator shall not occur at beginning or end of a macro

The '##' (token concatenation) operator is used to paste together adjacent preprocessor tokens, so it cannot be used at the beginning or end of a macro body.

W 86 escape character truncated to 8 bit value

The value of a hexadicimal escape sequence (a backslash, \, followed by a 'x' and a number) must fit in 8 bits storage. The number of bits per character may not be greater than 8. The following is an example of this warning:

```
char c = '\xabc';      /* error */
```

E 87 concatenated string too long

The resulting string was longer than the limit of 1500 characters.

W 88 "*name*" redeclared with different linkage

The specified identifier was already declared. This warning is issued when you try to redeclare an object with a different basic storage class, and both objects are not declared extern or static. The following is an example of this warning:

```
int i;
int i();                /* error E 64 and warning */
```

E 89 illegal bitfield declarator

A bit field may only be declared as an integer, not as a pointer or a function for example. So, `struct {int *a:1;} s;` is not allowed.

E 90 #error *message*

The *message* is the descriptive text supplied in a '#error' preprocessor directive.

W 91 no prototype for function "*name*"

Each function should have a valid function prototype.

W 92 no prototype for indirect function call

Each function should have a valid function prototype.

I 94 hiding earlier one

Additional message which is preceded by error E 63. The second declaration will be used.

F 95 protection error: *message*

Something went wrong with the protection key initialization. The message could be: "Key is not present or printer is not correct.", "Can't read key.", "Can't initialize key.", or "Can't set key–model".

E 96 syntax error in #define

`#define id(` requires a right–parenthesis ')'.

E 97 "..." incompatible with old–style prototype

If one function has a parameter type list and another function, with the same name, is an old–style declaration, the parameter list may not have ellipsis. The following is an example of this error:

```
int f(int, ...);
int f();                    /* error, old-style */
```

E 98 function type cannot be inherited from a typedef

A `typedef` cannot be used for a function definition. The following is an example of this error:

```
typedef int INTFN();
INTFN f {return (0);}     /* error */
```

F 99 conditional directives nested too deep

'#if', '#ifdef' or '#ifndef' directives may not be nested deeper than 50 levels.

E 100 +case or default label not inside switch

The `case:` or `default:` label may only appear inside a `switch`.

E 101 vacuous declaration

Something is missing in the declaration. The declaration could be empty or an incomplete statement was found. You must declare array declarators and `struct`, `union`, or `enum` members. The following are examples of this error:

```
int ;  /* error */

static int a[2] = { };    /* error */
```

E 102 +duplicate case or default label

Switch **case** values must be distinct after evaluation and there may be at most one **default:** label inside a **switch**.

E 103 may not subtract pointer from scalar

The only operands allowed on subtraction of pointers is pointer – pointer, or pointer – scalar. So, scalar – pointer is not allowed. The following is an example of this error:

```
int i;
int *pi = &i;
ff(1 - pi);          /* error */
```

E 104 left operand of *operator* has not struct/union type

The first operand of a '**.**' or '**->**' must have a **struct** or **union** type.

E 105 zero or negative array size – ignored

Array bound constants must be greater than zero. So, **char a[0];** is not allowed.

E 106 different constructors

Compatible function types with parameter type lists must agree in number of parameters and in use of ellipsis. Also, the corresponding parameters must have compatible types. This error is usually followed by informational message I 111. The following is an example of this error:

```
int f(int);
int f(int, int);    /* error different
                          parameter list */
```

E 107 different array sizes

Corresponding array parameters of compatible function types must have the same size.This error is usually followed by informational message I 111. The following is an example of this error:

```
int f(int [][2]);
int f(int [][3]);   /* error */
```

E 108 different types

Corresponding parameters must have compatible types and the type of each prototype parameter must be compatible with the widened definition parameter. This error is usually followed by informational message I 111. The following is an example of this error:

```
int f(int);
int f(long);              /* error different type
                               in parameter list */
```

E 109 floating point constant out of valid range

A floating point constant must have a value that fits in the type to which it was assigned. See section *Data Types* for the valid range of a floating point constant. The following is an example of this error:

```
float d = 10E9999;      /* error, too big */
```

E 110 function cannot return arrays or functions

A function may not have a return type that is of type array or function. A pointer to a function is allowed. The following are examples of this error:

```
typedef int F(); F f();  /* error */

typedef int A[2]; A g(); /* error */
```

I 111 parameter list does not match earlier prototype

Check the parameter list or adjust the prototype. The number and type of parameters must match. This message is preceded by error E 106, E 107 or E 108.

E 112 parameter declaration must include identifier

If the declarator is a prototype, the declaration of each parameter must include an identifier. Also, an identifier declared as a **typedef** name cannot be a parameter name. The following are examples of this error:

```
int f(int g, int) {return (g);} /* error */

typedef int int_type;
int h(int_type) {return (0);}   /* error */
```

E 114 incomplete struct/union type

The **struct** or **union** type must be known before you can use it. The following is an example of this error:

```
extern struct unknown sa, sb;
sa = sb;            /* 'unknown' does not have a
                          defined type */
```

The left side of an assignment (the lvalue) must be modifiable.

E 115 label "*name*" undefined

A **goto** statement was found, but the specified label did not exist in the same function or module. The following is an example of this error:

```
f1() { a: ; }         /* W 116 */
f2() { goto a; }      /* error, label 'a:' is
                           not defined in f2() */
```

W 116 label "*name*" not referenced

The given label was defined but never referenced. The reference of the label must be within the same function or module. The following is an example of this warning:

```
f() { a: ; }   /* 'a' is not referenced */
```

E 117 "*name*" undefined

The specified identifier was not defined. A variable's type must be specified in a declaration before it can be used. This error can also be the result of a previous error. The following is an example of this error:

```
unknown i; /* error, 'unknown' undefined */
i = 1;      /* as a result, 'i' is also
                 undefined */
```

W 118 constant expression out of valid range

A constant expression used in a **case** label may not be too large. Also when converting a floating point value to an integer, the floating point constant may not be too large. This warning is usually preceded by error E 16 or E 109. The following is an example of this warning:

```
int i = 10E88;   /* error and warning */
```

E 119 cannot take 'sizeof' bitfield or void type

The size of a bit field or **void** type is not known. So, the size of it cannot be taken.

E 120 cannot take 'sizeof' function

The size of a function is not known. So, the size of it cannot be taken.

E 121 not a function declarator

This is not a valid function. This may be due to a previous error. The following is an example of this error:

```
int f() return 0;   /* missing '{ }' */
int g() { }      /* error, 'g' is not a
                        formal parameter and
                        therefore, this is not a
                        valid function declaration */
```

E 122 unnamed formal parameter

The parameter must have a valid name.

W 123 function should return something

A **return** in a non–**void** function must have an expression.

E 124 array cannot hold functions

An array of functions is not allowed.

E 125 +function cannot return anything

A **return** with an expression may not appear in a **void** function.

W 126 missing return (function "*name*")

A non–**void** function with a non–empty function body must have a **return** statement.

E 129 cannot initialize "*name*"

Declarators in the declarator list may not contain initializations. Also, an **extern** declaration may have no initializer. The following are examples of this error:

```
{ extern int i = 0; }      /* error */
int f( i ) int i=0;        /* error */
```

W 130 operands of *operator* are pointers to different types

Pointer operands of an operator or assignment ('='), must have the same type. For example, the following code generates this warning:

```
long *pl;
int  *pi = 0;
pl = pi;     /* warning */
```

E 131 bad operand type(s) of *operator*

The operator needs an operand of another type. The following is an example of this error:

```
int *pi;
pi += 1.;     /* error, pointer on left; needs
                 integral value on right */
```

W 132 value of variable "*name*" is undefined

This warning occurs if a variable is used before it is defined. For example, the following code generates this warning:

```
int a,b;
a = b;  /* warning, value of b unknown */
```

E 133 illegal struct/union member type

A function cannot be a member of a **struct** or **union**. Also, bit fields may only have type **int** or **unsigned**.

E 134 bitfield size out of range – set to 1

The bit field width may not be greater than the number of bits in the type and may not be negative. The following example generates this error:

```
struct i { unsigned i : 999; }; /* error */
```

W 135 statement not reached

The specified statement will never be executed. This is for example the case when statements are present after a **return**.

E 138 illegal function call

You cannot perform a function call on an object that is not a function. The following example generates this error:

```
int i, j;
j = i();     /* error, i is not a function */
```

E 139 *operator* cannot have aggregate type

The type name in a (cast) must be a scalar (not a `struct`, `union` or a pointer) and also the operand of a (cast) must be a scalar. The following are examples of this error:

```
static union ui {int a;} ui ;
ui = (union ui)9;      /* cannot cast to union */
ff( (int)ui );         /* cannot cast a union
                          to something else */
```

E 140 *type* cannot be applied to a register/bit/bitfield object or builtin/inline function

For example, the '&' operator (address) cannot be used on registers and bit fields. So, `func(&r6);` and `func(&bitf.a);` are invalid.

E 141 *operator* requires modifiable lvalue

The operand of the '++', or '––' operator and the left operand of an assignment or compound assignment (lvalue) must be modifiable. The following is an example of this error:

```
const int i = 1;
i = 3;      /* error, const cannot be
               modified */
```

E 143 too many initializers

There may be no more initializers than there are objects. The following is an example of this error:

```
static int a[1] = {1, 2};      /* error,
          only one object can be initialized */
```

W 144 enumerator "*name*" value out of range

An `enum` constant exceeded the limit for an `int`. The following is an example of this warning:

```
enum { A = INT_MAX, B };       /* warning,
            B does not fit in an int anymore */
```

E 145 requires enclosing curly braces

A complex initializer needs enclosing curly braces. For example, `int a[] = 2;` is not valid, but `int a[] = {2};` is.

E 146 argument *#number*: memory spaces do not match

With prototypes, the memory spaces of arguments must match.

W 147 argument *#number*: different levels of indirection

With prototypes, the types of arguments must be assignment compatible. The following code generates this warning:

```
int i; void func(int,int);
func( 1, &i );     /* warning, argument 2 */
```

W 148 argument *#number*: struct/union type does not match

With prototypes, both the prototyped function argument and the actual argument was a `struct` or `union`., but they have different tags. The tag types should match. The following is an example of this warning:

```
f(struct s); /* prototype */
main()
{
   struct { int i; } t;
   f( t ); /* t has other type than s */
}
```

E 149 object "*name*" has zero size

A `struct` or `union` may not have a member with an incomplete type. The following is an example of this error:

```
struct { struct unknown m; } s; /* error */
```

W 150 argument *#number*: pointers to different types

With prototypes, the pointer types of arguments must be compatible. The following example generates this warning:

```
int f(int*);
long *l;
f(l);     /* warning */
```

W 151 ignoring memory specifier

Memory specifiers for a `struct`, `union` or `enum` are ignored.

E 152 operands of *operator* are not pointing to the same memory space

Be sure the operands point to the same memory space. This error occurs, for example, when you try to assign a pointer to a pointer from a different memory space.

E 153 'sizeof' zero sized object

An implicit or explicit **sizeof** operation references an object with an unkown size. This error is usually preceded by error E 119 or E 120, cannot take 'sizeof'.

E 154 argument *#number*: struct/union mismatch

With prototypes, only one of the prototyped function argument or the actual argument was a **struct** or **union**. The types should match. The following is an example of this error:

```
f(struct s); /* prototype */

main()
{
   int i;
   f( i ); /* i is not a struct */
}
```

E 155 casting lvalue '*type*' to '*type*' is not allowed

The operand of the '++', or '––' operator or the left operand of an assignment or compound assignment (lvalue) may not be cast to another type. The following is an example of this error:

```
int i = 3;
++(unsigned)i;      /* error, cast expression
                       is not an lvalue */
```

E 157 "*name*" is not a formal parameter

If a declarator has an identifier list, only its identifiers may appear in the declarator list. The following is an example of this error:

```
int f( i ) int a;   /* error */
```

E 158 right side of *operator* is not a member of the designated struct/union

The second operand of '.' or '–>' must be a member of the designated **struct** or **union**.

E 160 pointer mismatch at *operator*

Both operands of *operator* must be a valid pointer. The following example generates this error:

```
int *pi = 44;       /* right side not a pointer */
```

E 161 aggregates around *operator* do not match

The contents of the structs, unions or arrays on both sides of the *operator* must be the same. The following example causes this error:

```
struct {int a; int b;} s;
struct {int c; int d; int e;} t;
s = t;      /* error */
```

E 162 *operator* requires an lvalue or function designator

The '&' (address) operator requires an lvalue or function designator. The following is an example of this error:

```
int i;
i = &( i = 0 );
```

W 163 operands of *operator* have different level of indirection

The types of pointers or addresses of the operator must be assignment compatible. The following is an example of this warning:

```
char **a;
char *b;
a = b;      /* warning */
```

E 164 operands of *operator* may not have type 'pointer to void'

The operands of *operator* may not have operand `(void *)`.

W 165 operands of *operator* are incompatible: pointer vs. pointer to array

The types of pointers or addresses of the operator must be assignment compatible. A pointer cannot be assigned to a pointer to array. The following is an example of this warning:

```
main()
{
      typedef int array[10];
      array a;
      array *ap = a;      /* warning */
}
```

E 166 *operator* cannot make something out of nothing

Casting type `void` to something else is not allowed. The following example generates this error:

```
void f(void);
main()
{
     int i;

     i = (int)f();     /* error */
}
```

E 170 recursive expansion of inline function "*name*"

An **_inline** function may not be recursive. The following example generates this error:

```
_inline int a (int i)
{
     a(i);     /* recursive call */
     return i;
}
main()
{
     a(1);     /* error */
}
```

E 171 +too much tail–recursion in inline function "*name*"

If the function level is greater than or equal to 40 this error is given. The following example generates this error:

```
_inline void a ()
{
     a();
}
main()
{
     a();
}
```

W 172 adjacent strings have different types

When concatenating two strings, they must have the same type. The following example generates this warning:

```
char b[] = L"abc""def";     /* strings have
                               different types */
```

E 173 'void' function argument

A function may not have an argument with type **void**.

E 174 not an address constant

A constant address was expected. Unlike a static variable, an automatic variable does not have a fixed memory location and therefore, the address of an automatic is not a constant. The following is an example of this error:

```
int *a;
static int *b = a; /* error */
```

E 175 not an arithmetic constant

In a constant expression no assignment operators, no '++' operator, no '––' operator and no functions are allowed. The following is an example of this error:

```
int a;
static int b = a++; /* error */
```

E 176 address of automatic is not a constant

Unlike a static variable, an automatic variable does not have a fixed memory location and therefore, the address of an automatic is not a constant. The following is an example of this error:

```
int a;     /* automatic */
static int *b = &a; /* error */
```

W 177 static variable "*name*" not used

A static variable is declared which is never used. To eliminate this warning remove the unused variable.

W 178 static function "*name*" not used

A static function is declared which is never called. To eliminate this warning remove the unused function.

E 179 +inline function "*name*" is not defined

Possibly only the prototype of the inline function was present, but the actual inline function was not. The following is an example of this error:

```
_inline int a(void);     /* prototype */

main()
{
     int b;
     b = a();              /* error */
};
```

E 180 illegal target memory (*memory*) for pointer

The pointer may not point to *memory*. For example, a pointer to bitaddressable memory is not allowed.

E 181 invalid cast to function

A cast to type function is not allowed. A cast to a function pointer type is allowed.

W 182 argument *#number*: different types

With prototypes, the types of arguments must be compatible.

W 183 variable '*name*' possibly uninitialized

Possibly an initialization statement is not reached, while a function should return something. The following is an example of this warning:

```
int a;

int f(void)
{
   int i;

   if ( a )
   {
   i = 0; /* statement not reached */
   }
   return i; /* warning */
}
```

W 184 empty pragma name in –z option – ignored

The **–z** option requires a pragma name.

I 185 (prototype synthesized at line *number* in "*name*")

This is an informational message containing the source file position where an old–style prototype was synthesized. This message is preceded by error E 146, W 147, W 148, W 150, E 154, W 182 or E 203.

E 186 array of type bit is not allowed

An array cannot contain bit type variables.

E 187 illegal structure definition

A structure can only be defined (initialized) if its members are known. So, `struct unknown s = { 0 };` is not allowed.

E 188 structure containing bit–type fields is forced into bitaddressable area

This error occurs when you use a bitaddressable storage type for a structure containing bit–type members.

E 189 pointer is forced to bitaddressable, pointer to bitaddressable is illegal

A pointer to bitaddressable memory is not allowed.

W 190 ”long float” changed to ”float”

In ANSI C floating point constants are treated having type `double`, unless the constant has the suffix 'f'. If you have specified an option to use float constants, a `long` floating point constant such as `123.12fl` is changed to a `float`.

E 191 recursive struct/union definition

A `struct` or `union` cannot contain itself. The following example generates this error:

```
struct s { struct s a; } b;      /* error */
```

E 192 missing filename after –f option

The **–f** option requires a filename argument.

E 194 cannot initialize typedef

You cannot assign a value to a typedef variable. So, `typedef i=2;` is not allowed.

W 195 constant expression out of range –– truncated

The resulting constant expression is too large to fit in the specified data type. The value is truncated. The following example generates this warning:

```
int i = 140000L;      /* warning, value is too large
                         to fit in an int */
```

W 196 constant expression out of range due to signed/unsigned type mismatch

The resulting constant expression is too large to fit in the specified data type. The following example generates this warning:

```
int i = 40000U; /* the unsigned value is too large
                   to fit in a signed int */
                /* unsigned int i = 40000U; is OK */
```

W 197 unrecognized –w argument: *argument*

The **–w** option only accepts a warning number or the text 'strict' as an argument. See the description of the **–w** option for details.

W 198 trigraph sequence replaced

Trigraphs are used in the C language to create special characters on obsolete terminals with a limited character set. When they are replaced in your source, e.g. in a string, they may give rise to very obscure errors.

F 199 demonstration package limits exceeded

The demonstration package has certain limits which are not present in the full version. Contact TASKING for a full version.

W 200 unknown pragma – ignored

The compiler ignores pragmas that are not known. For example, `#pragma unknown`.

W 201 *name* cannot have storage type – ignored

A `register` variable or an automatic/parameter cannot have a storage type. To eliminate this warning, remove the storage type or place the variable outside a function.

E 202 '*name*' is declared with 'void' parameter list

You cannot call a function with an argument when the function does not accept any (`void` parameter list). The following is an example of this error:

```
int f(void);     /* void parameter list */

main()
{
     int i;
     i = f(i);    /* error */
     i = f();     /* OK */
}
```

E 203 too many/few actual parameters

With prototyping, the number of arguments of a function must agree with the protoype of the function. The following is an example of this error:

```
int f(int);     /* one parameter */

main()
{
     int i;
     i = f(i,i);    /* error, one too many */
     i = f(i);      /* OK */
}
```

W 204 U suffix not allowed on floating constant – ignored

A floating point constant cannot have a 'U' or 'u' suffix.

W 205 F suffix not allowed on integer constant – ignored

An integer constant cannot have a 'F' or 'f' suffix.

E 206 '*name*' named bit–field cannot have 0 width

A bit field must be an integral contstant expression with a value greater than zero.

E 207 list of rule numbers expected after "–misrac" option

Add the numbers of the MISRA C rules to the –misrac option to specifiy the rules that must be checked. See Appendix B *MISRA C*

W 208 unsupported MISRA C rule number *number*.

Specified MISRA C rule number is not supported.

E 209 +MISRA C rule *number* violation: *rule_description*

A specified MISRA C rule is violated.

E 212 "*name*": missing static function definition

A function with a `static` prototype misses its definition.

W 213 invalid string/character constant in non–active part of source

This part of the source is skipped.

E 214 second occurrence of #pragma asm or asm_noflush

`#pragma asm`/`#pragma endasm` blocks cannot be nested. Use `#pragma endasm` before starting a new `#pragma asm`/`#pragma endasm` block.

E 215 "#pragma endasm" without a "#pragma asm"

A `#pragma endasm` must always have a corresponding `#pragma asm` or `#pragma asm_noflush`.

W 216 suggest parentheses around assignment used as truth value

Generated when the argument of an `if` statement is actually an assignment (might indicate a typing error).

In the example below W 216 will be generated because of a suspicious assignment within an `if` condition.

```
int func( int a, int b, int c )
{
     if ( a = b )
     {
          return c;
     }
     return a;
}
```

W 225 dereferencing void pointer

A void pointer cannot be dereferenced. The following is an example of this warning:

```
volatile void * p;

void f(void)
{
   *p;
   return;
}
```

W 227 MISRA C rule *number* violation: *rule*

F 228 MISRA C rule *number* violation: *rule*

A specified MISRA C rule is violated.

W 303 variable '*name*' uninitialized

Possibly an initialization statement is not reached, while a function should return something. The following is an example of this warning:

```
int a;

int f(void)
{
   int i;

   if ( a )
   {
   i = 0; /* statement not reached */
   }
   return i; /* warning */
}
```

E 327 too many arguments to pass in registers for _asmfunc '*name*'

An **_asmfunc** function uses a fixed register–based interface between C and assembly, but the number of arguments that can be passed is limited by the number of available registers. With function *name* this limit was reached.

Backend

W 501 initializer was truncated

When a value does not fit in a character, structure or integer, the value is truncated.

E 502 fail to generate code for *type*

The compiler could not generate code for this type.

F 504 object doesn't fit in memory: *memory*

The object is too large for the specified memory type.

E 519 no indirection allowed on bit type

Bitaddressable memory is only direct addressable. Pointer to `_bit` and array of `_bit` are not allowed.

E 521 out of temporary bit storage, simplify expression

The expression used too many static bit temporaries.

E 527 move to read–only field

Of course you can only read from a read–only field.

E 531 restriction: impossible to convert to '*type*'

A structure or union cannot be converted to type `bit`, `char`, `float`, `int` or `long`.

E 539 *operator* not allowed on *type* type

Some operators are not allowed on type `_bit`.

E 540 "*function*" is not a 'plmprocedure'

You cannot write plmprocedures within 'C', only a call is possible.

E 541 not allowed to switch on pointer type

A pointer type is not allowed as switch expression.

E 542 switch only possible on char/int/long type

A `_bit` typed expression is not allowed as switch expression.

E 543 static model: non–register parameters not allowed with function pointer

Static passing does not allow parameters with function pointers.

E 544 illegal testclear argument

See the description of the intrinsic function `testclear` for the correct syntax.

W 545 no address available for variable argument list

There is no address reference available for `va_start`.

E 547 calling an interrupt routine is not allowed

It is never allowed to call an interrupt function.

E 550 assignment/parameter/return not allowed with bit–structure

A _bit structure can only contain members of type `_bit`.

F 551 illegal bank number

The register bank number must be a number in the range 0 to 3.

F 552 illegal rom model

Only the rom models 's', 'm' and 'l' are allowed.

F 553 illegal memory model

Only the memory models 's', 'a', 'l' and 'r' are allowed.

F 554 illegal memory type specified

See the description of the **–R** option or the **–m** option for the correct syntax.

F 555 illegal memory size given

The argument of the **–m** option can only contain numerical values

W 556 _plmprocedure is in conflict with _regparm

`_regparm` and `_plmprocedure` cannot be used together.

W 558 static model (overlaying) disables –rm (rom medium) –ignored

The **–rm** option is useless in the static model because each function has its own code segment.

E 559 impossible to save structure result, simplify expression

The structure or union could not be saved on stack.

W 560 reentrant interrupt function ("*name*") with local variables: adjust library

Use the protected version of the library.

W 561 interrupt function calling a reentrant function: adjust library

A reentrant function cannot be called from an interrupt function.

E 562 bit–type parameter/automatic only allowed in static models

`_bit` parameters/automatics are not allowed in `_reentrant` functions.

W 563 automatic cannot have storage type rom – ignored

It is not possible to store automatic variables in rom. The rom type is removed.

E 564 '*name*' is illegal memory for function

A function can have return type `_bit` or program code. A function cannot have return type `_bitbyte`.

W 565 conversion of long address to short address

The conversion of xdat to pdat pointer is allowed: same physical space.

F 566 illegal number in *option* option

The argument of the **–a**, **–b**, **–c** and **–x** option can only contain numerical values.

E 570 Cannot take address of bit–variable or bit–structure

You cannot take the address of a bit–variable or bit–structure.

W 575 'reg'–field (CSE–administration : *number*) not empty

The CSE administation is cleaned up and compilation continues.

E 576 _at() expects a constant address

You can only use an expression that evaluates to a numerical address.

E 577 _at() address out of range for this type of object

The absolute address is not present in the specified memory space.

E 578 _at() only valid for global variables

Only global variables can be placed on absolute addresses.

E 579 'bitoffset' for '*name*' must be a constant value between 0 and *number*

The bitoffset wihin a bitbyte must be a constant value between 0 and (*size_of_bitbyte* * 8) – 1.

E 580 specified object not bitaddressable

`_bitbyte` can only be used on bitaddressable memory. Also, the SFR must be bitaddressable.

E 581 different register bank ('using')

The register bank numbers of the calling and called function must be the same.

E 583 _at() only allowed on non initialized variables

Absolute variables cannot be initialized.

W 585 duplicate function qualifier – 'interrupt(*number*)/ using(*number*)/ plmprocedure()' ignored

Only one function qualifier is allowed.

W 586 R2/R3 contained a CSE, which could have been used once more

The CSE in the R2 or R3 register can be used once more.

W 587 '*number*' illegal *name* number (0 to *max*) – ignored

An `_interrupt` number must be a number between 0 and 16. An `_using` number must be a number between 0 and 3.

E 589 interrupt function must have void result and void parameter list

A function declared with `_interrupt(n)` is not allowed to have any arguments and must not return anything.

E 591 conflict in 'interrupt' attribute

The attributes of the current function qualifier declaration and the previous function qualifier declaration are not the same.

E 592 different 'interrupt/using/plmprocedure' number

The interrupt/using/plmprocedure number of the current function qualifier declaration and the previous function qualifier declaration are not the same.

W 593 function qualifier used with non–function

A function qualifier can only be used on functions.

W 594 duplicate or conflicting function qualifier – '*name*' ignored

Only one function qualifier is allowed.

W 595 _at() has no effect on external declaration

When declared **extern** the variable is not allocated by the compiler.

E 596 function models (_small/_aux/_large/_reentrant) do not match

The function and the prototype of the function must have the same model qualifier.

E 597 parameter passing attributes (_regparm/_cdecl) do not match

The function and the prototype of the function must have the same parameter passing attributes.

E 598 _atbit() only possible on objects, not on constant addresses

_atbit() is not possible on constant addresses, they are only possible on **_bitbyte** or **_sfrbyte** objects.

E 599 _atbit() only possible on _bitbyte/_sfrbyte objects

_atbit() only accepts **_bitbyte** or **_sfrbyte** objects as an argument.

E 600 _atbit() only possible for _bit/_sfrbit objects

Only **_bit** and **_sfrbit** objects can be declared with **_atbit()**.

E 601 _atbit() object must have same storage as target object

The storage class of both **_atbit()** objects must be the same.

E 602 _sfrbit object can only have _atbit() on an _sfrbyte object
_bit object can only have _atbit() on a _bitbyte object

You cannot specify a **_sfrbit** object with **_atbit()** on a **_bitbyte** object, and you cannot specify a **_bit** object with **_atbit()** on a **_sfrbyte** object.

E 603 in space _bdat only integral objects are allowed

Space **_bdat** is bitaddressable ram. In this space you can only use integral objects.

E 604 illegal interrupt option, specify –ivo=<value>

Only the **–ivo** option is allowed. Check the syntax of your **–ivo** option.

E 605 illegal interrupt vector option, specify –v or –vf

Only the **–v** or **–vf** option are allowed. Check the syntax of your **–v** option.

E 606 unknown register name: "*name*"

You specified a non–existing register name to pragma **intsave**. Correct the name.

E 607 register name expected

Pragma **intsave** requires a register name.

E 608 '_frame()' without '_interrupt()'

The **_frame** function qualifier can only be used on **_interrupt** functions.

E 609 different _frame() lists

The list of registers/SFRs of the current **_frame** function qualifier declaration and the previous **_frame** function qualifier declaration are not the same.

E 611 code generation attribute _noregaddr does not match

The attribute **_noregaddr** does not match with the prototype of the function.

E 612 _inline useless on interrupt function

Interrupt functions cannot be defined as **_inline** functions.

E 613 _sfrbit/byte only allowed for global variables

Only global variables can be placed on absolute addresses.

F 614 code generation stopped

The compiler found an unresolvable error and cannot continue.

E 617 _atbit() not possible on *type*: "*name*"

You cannot use: struct / union members, tags, labels, parameters or inline function locals as a base symbol to define bits in.

W 619 interrupt uses default register bank

The interrupt function was specified to use the default register bank (usually register bank 0). For example the following may result in a run–time error if 0 is the default register bank:

```
void _interrupt(1) _using(0) ISR(void);
```

Choose another register bank for the interrupt function.

E 622 interrupt address *hexvalue* already used for function *name*

Each interrupt function must use a unique interrupt address.

E 627 _push()/_pop(): expecting SFR or constant parameter

The `_push()` and `_pop()` intrinsic functions require an SFR or a constant address parameter, e.g. `_push(ACC)` or `_pop(0x80)`.

E 628 _jmp(): expecting function identifier

The `_jmp()` intrinsic function requires a function identifier argument.

F 629 *option* has been deprecated

The specified option is no longer supported.

W 630 probable endianness mismatch in _sfrword *sfr* definition

There is a possible endianness mismatch in the `_sfrword` definition for *sfr* and the corresponding low and hight byte `_sfrbyte` definitions. Probably you should use the `_little` specifier on the `_sfrword` declaration.

W 632 inconsistent _sfrbyte definitions for _sfrword *sfr*

There is a possible inconsistency in the naming of the low and/or high byte `_sfrbyte` corresponding to `_sfrword` *sfr*.

CHAPTER 6

LIBRARIES

TASKING

9 CHAPTER

6.1 INTRODUCTION

This chapter describes the library functions delivered with the compiler. Some functions (e.g. `printf()`, `scanf()`) can be edited to match your needs. **cc51** comes with libraries in object format per memory model and with header files containing the appropriate prototype of the library functions. The library functions are also shipped in source code (C or assembly).

A number of standard operations within C are too complex to generate inline code for (e.g. 16 bit signed divide). These operations are implemented as run–time library functions. The run–time library routines are added to the C library.

6.2 HEADER FILES

The following header files are delivered with the C compiler:

<**assert.h**> assert

<**cc51.h**> Special file with **cc51** definitions. No C functions.

<**ctype.h**> isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii, _tolower, tolower, _toupper, toupper

<**errno.h**> Error numbers. No C functions.

<**float.h**> Constants related to floating point arithmetic.

<**keil.h**> Support for migration from Keil C–51 to TASKING C–51**.**

<**limits.h**> Limits and sizes of integral types. No C functions.

<**math.h**> acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan, tanh

<**setjmp.h**> longjmp, setjmp

<**simio.h**> _simi, _simo

<**stdarg.h**> va_arg, va_end, va_start

<**stddef.h**> offsetof, definition of special types.

<**stdio.h**> fgetc, fgets, fprintf, fputc, fputs, fread, fscanf, fwrite, getc, getchar, gets, _ioread, _iowrite, printf, putc, putchar, puts, scanf, sprintf, sscanf, ungetc, vfprintf, vprintf, vsprintf

<**stdlib.h**> abs, atof, atoi, atol, bsearch, calloc, div, exit, free, labs, ldiv, malloc, qsort, rand, realloc, strtod, strtol, strtoul, srand

<**string.h**> memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok, ididcpy, ididmove, idxdcpy, idxdmove, romidcpy, romidmove, romxdcpy, romxdmove, xdidcpy, xdidmove, xdxdcpy, xdxdmove

<**time.h**> clock, time. All functions are delivered as skeletons.

6.3 C LIBRARIES

The C library contains C library functions. All C library functions are described in this chapter. These functions are only called by explicit function calls in your application program.

The C library uses the following name syntax:

Compiler Model		Library to link	
		Non–protected	Protected
Small (default)	(–Ms)	c51s.lib	c51sp.lib
Small for 751 derivative	(–Ms –rs)	c751s.lib	c751sp.lib
Auxpage	(–Ma)	c51a.lib	c51ap.lib
Large	(–Ml)	c51l.lib	c51lp.lib
Reentrant	(–Mr)	c51r.lib	c51rp.lib

Table 6–1: C library name syntax

The `c751s.lib` library is made especially for the 80751/80752 derivatives, (containing a small amount of ROM, no extern RAM is possible). All routines which use `pdat` or `xdat` variables are removed from this library.

Use the protected libraries to prevent the occurrence of interrupts while updating the virtual stack pointer.

In EDE you can select a protected library by enabling the option `Use protection on virtual stack pointer updates` in the `Linker | Stack/Heap` entry of the `Project | Project Options...` dialog.

Description	Library to link
For all models:	
Floating Point Library (needs external RAM)	float.lib
Floating Point Library (basic fp operations, no external RAM neeeded)	floats.lib

Table 6–2: Floating point library name syntax

See section 6.3.3, *Printf and Scanf Formatting Routines*, for the name syntax of the delivered printf and scanf libraries.

See section 7.8, *Multiple Data Pointer Support* in chapter *Run–time Environment*, for the name syntax of the multiple data pointer libraries.

When you use a derivative without external RAM (like the 80751/80752) you cannot use the default floating point library `float.lib`, since that library uses external RAM. In that case use the floating point library `floats.lib` instead.

When you use floating point, the library `float.lib` must always be the last library linked, it should be placed after the, C library. For further explanation how to link your application, see section 7.11 *Linking an Application* in chapter *Run–time Environment*.

The floating point library `floats.lib` does not need any external RAM. When you use this library you have to change the startup code to place the floating point stack in internal RAM. Arithmetic routines link `sin()`, `cos()`, etc. are not present in this library, only basic floating point operations can be done.

Using the reentrant model, you might need to change some routines. See section 7.10, *Reentrant Model / _reentrant Functions* in chapter *Run–time Environment.*

In order to allow the library functions to use a NULL pointer as an illegal address, you should avoid memory location `xdat:0` to be referred to by a pointer. You can achieve this by either using the linker RESERVE control (e.g. reserve 1 byte: `RESERVE(xdata(0,0))`), or be sure that this memory location is occupied by a plain (static) variable (which is not pointed to by a pointer).

In EDE you can prevent pointers on address 0 by enabling the option `Reserve first byte of XDATA to prevent pointers on address0` in the `Linker | Reserved Areas` entry of the `Project | Project Options...` dialog.

6.3.1 C LIBRARY IMPLEMENTATION DETAILS

A detailed description of the delivered C library is shown in the following list.

Explanation :

- Y – Fully implemented
- I – Implemented, but need some user written low level routine
- L – Delivered as a skeleton

File	Imple–mented	Routine name	Description / Reason
assert.h	Y	'assert()' macro	Macro definition
ctype.h	Y		Most of the routines are delivered as macro AND as function (as prescribed by ANSI).
	Y	isalnum	
	Y	isalpha	
	Y	iscntrl	
	Y	isdigit	
	Y	isgraph	
	Y	islower	
	Y	isprint	
	Y	ispunct	
	Y	isspace	
	Y	isupper	
	Y	isxdigit	
	Y	tolower	
	Y	toupper	
	Y	_tolower	Not defined by ANSI
	Y	_toupper	Not defined by ANSI
	Y	isascii	Not defined by ANSI
	Y	toascii	Not defined by ANSI
errno.h	Y		Only Macros
float.h	Y		
limits.h	Y		Only Macros

File	Imple-mented	Routine name	Description / Reason
math.h	Y		
	Y	acos	
	Y	asin	
	Y	atan	
	Y	atan2	
	Y	ceil	
	Y	cos	
	Y	cosh	
	Y	exp	
	Y	fabs	
	Y	floor	
	Y	fmod	
	Y	frexp	
	Y	ldexp	
	Y	log	
	Y	log10	
	Y	modf	
	Y	pow	
	Y	sin	
	Y	sinh	
	Y	sqrt	
	Y	tan	
	Y	tanh	
setjmp.h	Y		
	Y	longjmp	
	Y	setjmp	
stdarg.h	Y		
	Y	va_arg	
	Y	va_end	
	Y	va_start	
stddef.h	Y		Only Macros
stdio.h	Y		Due to 'stdarg.h/varargs.h' conflicts, the routines 'vprintf()', 'vfprintf()', 'vsprintf()' are not ANSI yet.
	I	fgetc	Needs _ioread
	I	fgets	Needs _ioread
	I	fprintf	Needs _iowrite
	I	fputc	Needs _iowrite
	I	fputs	Needs _iowrite
	I	fread	Needs _ioread
	I	fscanf	Needs _ioread
	I	fwrite	Needs _iowrite

File	Imple–mented	Routine name	Description / Reason
	I	getc	Needs _ioread
	I	getchar	Needs _ioread
	I	gets	Needs _ioread
	I	printf	Needs _iowrite
	I	putc	Needs _iowrite
	I	putchar	Needs _iowrite
	I	puts	Needs _iowrite
	I	scanf	Needs _ioread
	Y	sprintf	
	Y	sscanf	
	Y	ungetc	
	I	vfprintf	Needs _iowrite
	I	vprintf	Needs _iowrite
	Y	vsprintf	
	I	_ioread	Low level input routine
	I	_iowrite	Low level output routine
stdlib.h	Y		
	Y	abs	
	Y	atof	
	Y	atoi	
	Y	atol	
	Y	bsearch	
	Y	calloc	
	Y	div	
	Y	exit	Calls __EXIT in cstart
	Y	free	
	Y	labs	
	Y	ldiv	
	Y	malloc	
	Y	qsort	
	Y	rand	
	Y	realloc	
	Y	strtod	
	Y	strtol	
	Y	strtoul	
	Y	srand	

File	Imple-mented	Routine name	Description / Reason
string.h	Y		
	Y	memchr	
	Y	memcmp	
	Y	memcpy	
	Y	memmove	
	Y	memset	
	Y	strcat	
	Y	strchr	
	Y	strcmp	
	Y	strcpy	
	Y	strcspn	
	Y	strlen	
	Y	strncat	
	Y	strncmp	
	Y	strncpy	
	Y	strpbrk	
	Y	strrchr	
	Y	strspn	
	Y	strstr	
	Y	strtok	
	Y	ididcpy	Not defined by ANSI
	Y	ididmove	Not defined by ANSI
	Y	idxdcpy	Not defined by ANSI
	Y	idxdmove	Not defined by ANSI
	Y	romidcpy	Not defined by ANSI
	Y	romidmove	Not defined by ANSI
	Y	romxdcpy	Not defined by ANSI
	Y	romxdmove	Not defined by ANSI
	Y	xdidcpy	Not defined by ANSI
	Y	xdidmove	Not defined by ANSI
	Y	xdxdcpy	Not defined by ANSI
	Y	xdxdmove	Not defined by ANSI
time.h	Y		real time clock not supported
	I	clock	Uses SFRs 0xC1 to 0xC4
	I	time	

6.3.2 C LIBRARY INTERFACE DESCRIPTION

_ioread

```
#include <stdio.h>
_regparm int _ioread( FILE *fp );
```

Low level input function. The delivered library contains a call to "_simi()". To perform real I/O, you must customize this function. _ioread is used by all input functions (scanf, getc, gets, etc.). See the file **_ioread.c** in the **lib\src** directory demonstrating an example implementation of this low level input function using the CrossView Pro I/O Simulation input feature.

_iowrite

```
#include <stdio.h>
_regparm int _iowrite( int c, FILE *fp );
```

Low level output function. The delivered library contains a call to "_simo()". To perform real I/O, you must customize this function. _iowrite is used by all output functions (printf, putc, puts, etc.). See the file **_iowrite.c** in the **lib\src** directory demonstrating an example implementation of this low level output function using the CrossView Pro I/O Simulation output feature.

_simi

```
#include <simio.h>
void _regparm
_simi( unsigned char stream, char *port,
       unsigned char len );
```

CrossView Pro I/O Simulation input interface function.

See also "_ioread()".

_simo

```
#include <simio.h>
void _regparm
_simo( unsigned char stream, char *port,
       unsigned char len );
```

CrossView Pro I/O Simulation output interface function.

See also "_iowrite()".

_tolower

```
#include <ctype.h>
_regparm int _tolower( int c );
```

Converts **c** to a lowercase character, does not check if **c** really is an uppercase character. This is a non–ANSI function.

Returns the converted character.

_toupper

```
#include <ctype.h>
_regparm int _toupper( int c );
```

Converts **c** to an uppercase character, does not check if **c** really is a lowercase character. This is a non–ANSI function.

Returns the converted character.

abs

```
#include <stdlib.h>
_regparm int abs( int n );
```

Returns the absolute value of the signed int argument.

acos

```
#include <math.h>
_regparm double acos( double x );
```

Returns the arccosine $\cos^{-1}(x)$ of `x` in the range $[0, \pi]$, `x` $\in [-1, 1]$.

asin

```
#include <math.h>
_regparm double asin( double x );
```

Returns the arcsine $\sin^{-1}(x)$ of `x` in the range $[-\pi/2, \pi/2]$, `x` $\in [-1, 1]$.

assert

```
#include <assert.h>
_regparm void assert( int expr );
```

When compiled with NDEBUG, this is an empty macro. When compiled without NDEBUG defined, it checks if `expr` is true. If it is true, then a line like:

`"Assertion failed: `*`expression`*`, file `*`filename`*`, line `*`num`*`"`

is printed.

Returns nothing.

atan

```
#include <math.h>
_regparm double atan( double x );
```

Returns the arctangent $\tan^{-1}(x)$ of `x` in the range $[-\pi/2, \pi/2]$. `x` $\in [-1, 1]$.

atan2

```
#include <math.h>
_regparm double atan2( double y, double x );
```

Returns the result of: $\tan^{-1}(y/x)$ in the range $[-\pi, \pi]$.

atof

```
#include <stdlib.h>
_regparm double atof( const char *s );
```

Converts the given string to a double value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the double value.

atoi

```
#include <stdlib.h>
_regparm int atoi( const char *s );
```

Converts the given string to an integer value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the integer value.

atol

```
#include <stdlib.h>
_regparm long atol( const char *s );
```

Converts the given string to a long value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the long value.

bsearch

```
#include <stdlib.h>
_regparm _reentrant void *bsearch(
  const void *key, const void *base, size_t n,
  size_t size, _regparm _reentrant int (* cmp)
  (const void *, const void *) );
```

This function searches in an array of `n` members, for the object pointed to by `ptr`. The initial base of the array is given by `base`. The size of each member is specified by `size`. The given array must be sorted in ascending order, according to the results of the function pointed to by `cmp`.

Returns a pointer to the matching member in the array, or NULL when not found.

calloc

```
#include <stdlib.h>
_regparm void xdat *calloc( size_t nobj,
                   size_t size );
```

The allocated space is filled with zeros. The maximum space that can be allocated can be changed by customizing the heap size (see section 7.5, *Heap*). By default no heap is allocated. When ”calloc()” is used while no heap is defined, the linker gives an error.

Be aware, for ’small’ and ’aux’ model, pointers to the allocated area must be declared as pointing to `xdat`.

Returns a pointer to space in external memory for `nobj` items of `size` bytes length.
NULL if there is not enough space left.

ceil

```
#include <math.h>
_regparm double ceil( double x );
```

Returns the smallest integer not less than `x`, as a double.

clock

```
#include <time.h>
clock_t clock( void );
```

Determines the processor time used.

Returns number of microseconds since the last reset, assuming a 12 MHz cpu.

cos

```
#include <math.h>
_regparm double cos( double x );
```

Returns the cosine of `x`.

cosh

```
#include <math.h>
_regparm double cosh( double x );
```

Returns the hyperbolic cosine of `x`.

div

```
#include <stdlib.h>
_regparm div_t div( int num, int denom );
```

Both arguments are integers. The returned quotient and remainder are also integers.

Returns a structure containing the quotient and remainder of `num` divided by `denom`.

exit

```
#include <stdlib.h>
_regparm void exit( int status );
```

Terminates the program normally. Acts as if 'main()' returns with `status` as the return value.

Returns zero, on successful termination.

exp

```
#include <math.h>
_regparm double exp( double x );
```

Returns the result of the exponential function e^x.

fabs

```
#include <math.h>
_regparm double fabs( double x );
```

Returns the absolute double value of `x`. $|x|$

fgetc

```
#include <stdio.h>
_regparm int fgetc( FILE *stream );
```

Reads one character from the given `stream`.

Returns the read character, or EOF on error.

fgets

```
#include <stdio.h>
_regparm char *fgets( char *s, int n,
          FILE *stream );
```

Reads at most the next `n`–1 characters from the given `stream` into the array `s` until a newline is found.

Returns `s`, or NULL on EOF or error.

floor

```
#include <math.h>
_regparm double floor( double x );
```

Returns the largest integer not greater than `x`, as a double.

fmod

```
#include <math.h>
_regparm double fmod( double x, double y );
```

Returns the floating–point remainder of `x/y`, with the same sign as `x`. If `y` is zero, the result is implementation–defined.

fprintf

```
#include <stdio.h>
_regparm int fprintf( FILE *stream,
                      const char *format, ... );
```

Performs a formatted write to the given `stream`.

See also ”printf()”, ”_iowrite()” and section 6.3.3, *Printf and Scanf Formatting Routines*.

fputc

```
#include <stdio.h>
_regparm int fputc( int c, FILE *stream );
```

Puts one character onto the given `stream`.

See also "_iowrite()".

Returns EOF on error.

fputs

```
#include <stdio.h>
_regparm int fputs( const char *s,
                    FILE *stream );
```

Writes the string to a `stream`. The terminating NULL character is not written.

See also "_iowrite()".

Returns NULL if successful, or EOF on error.

fread

```
#include <stdio.h>
_regparm size_t fread( void *ptr, size_t size,
                       size_t nobj, FILE *stream );
```

Reads `nobj` members of `size` bytes from the given `steam` into the array pointed to by `ptr`.

See also "_ioread()".

Returns the number of successfully read objects.

free

```
#include <stdlib.h>
_regparm void free( void xdat *p );
```

Deallocates the space pointed to by `p`. `p` Must point to space earlier allocated by a call to ”calloc()”, ”malloc()” or ”realloc()”. Otherwise the behavior is undefined.

See also ”calloc()”, ”malloc()” and ”realloc()”.

Returns nothing

frexp

```
#include <math.h>
_regparm double frexp( double x, int *exp );
```

Splits `x` into a normalized fraction in the interval [1/2, 1>, which is returned, and a power of 2, which is stored in `*exp`. If `x` is zero, both parts of the result are zero. For example: `frexp( 4.0, &var )` results in $0.5 \cdot 2^3$. The function returns 0.5, and 3 is stored in var.

Returns the normalized fraction.

fscanf

```
#include <stdio.h>
_regparm int fscanf( FILE *stream,
                const char *format, ... );
```

Performs a formatted read from the given `stream`.

See also ”scanf()”, ”_ioread()” and section 6.3.3, *Printf and Scanf Formatting Routines*.

Returns the number of items converted successfully.

fwrite

```
#include <stdio.h>
_regparm size_t fwrite( const void *ptr,
                        size_t size, size_t nobj,
                        FILE *stream );
```

Writes `nobj` members of `size` bytes to the given `stream` from the array pointed to by `ptr`.

Returns the number of successfully written objects.

getc

```
#include <stdio.h>
_regparm int getc( FILE *stream );
```

Reads one character out of the given `stream`. Currently #defined as getchar(), because FILE I/O is not supported.

See also "_ioread()".

Returns the character read or EOF on error.

getchar

```
#include <stdio.h>
_regparm int getchar( void );
```

Reads one character from standard input.

See also "_ioread()".

Returns the character read or EOF on error.

gets

```
#include <stdio.h>
_regparm char *gets( char *s );
```

Reads all characters from standard input until a newline is found. The newline is replaced by a NULL–character.

See also "_ioread()".

Returns a pointer to the read string or NULL on error.

ididcpy

```
#include <string.h>
_regparm void idat * ididcpy( idat void *cs,
                              idat void *ct,
                              size_t n );
```

Copies `n` bytes of data from `idat` memory to `idat` memory without worrying if data spaces overlap.

Returns a pointer to `ct`.

The compiler is able to handle different types of memory, i.e. `idat`, `data`, `pdat`, `xdat`, `const`. Sometimes it is necessary to have the possibility to copy data from one memory to another. Therefore the next functions have been created and placed in the C library :

ididcpy()	idxdcpy()
xdidcpy()	xdxdcpy()
romidcpy()	romxdcpy()
ididmove()	idxdmove()
xdidmove()	xdxdmove()
romidmove()	romxdmove()

The functions *xxxx*cpy() copy data from one memory to another memory without worrying if data spaces overlap. The functions *xxxx*move() first check if the data spaces overlap and then copy in the correct direction. When it is desired to copy to/from `data`, use the functions for `idat` (use a cast with the arguments). Moving data to `pdat` can be done by using the functions for `xdat` (use a cast with the arguments).

ididmove

```
#include <string.h>
_regparm void idat *ididmove( idat void *cs,
                              idat void *ct,
                              size_t n );
```

Moves `n` bytes of data from `idat` memory to `idat` memory. Overlapping spaces are handled correctly.

See also ”ididcpy()”.

idxdcpy

```
#include <string.h>
_regparm void xdat *idxdcpy( xdat void *cs,
                             const idat void *ct,
                             size_t n );
```

Copies `n` bytes of data from `idat` memory to `xdat` memory.

See also ”ididcpy()”.

idxdmove

```
#include <string.h>
_regparm void xdat *idxdmove( xdat void *cs,
                              const idat void *ct,
                              size_t n );
```

Moves `n` bytes of data from `idat` memory to `xdat` memory.

See also ”ididcpy()”.

isalnum

```
#include <ctype.h>
_regparm int isalnum( int c );
```

Returns a non–zero value when `c` is an alphabetic character or a number ([A–Z][a–z][0–9]).

isalpha

```
#include <ctype.h>
_regparm int isalpha( int c );
```

Returns a non–zero value when **c** is an alphabetic character ([A–Z][a–z]).

isascii

```
#include <ctype.h>
_regparm int isascii( int c );
```

Returns a non–zero value when **c** is in the range of 0 and 127. This is a non–ANSI function.

iscntrl

```
#include <ctype.h>
_regparm int iscntrl( int c );
```

Returns a non–zero value when **c** is a control character.

isdigit

```
#include <ctype.h>
_regparm int isdigit( int c );
```

Returns a non–zero value when **c** is a numeric character ([0–9]).

isgraph

```
#include <ctype.h>
_regparm int isgraph( int c );
```

Returns a non–zero value when **c** is printable, but not a space.

islower

```
#include <ctype.h>
_regparm int islower( int c );
```

Returns a non–zero value when **c** is a lowercase character ([a–z]).

isprint

```
#include <ctype.h>
_regparm int isprint( int c );
```

Returns a non–zero value when **c** is printable, including spaces.

ispunct

```
#include <ctype.h>
_regparm int ispunct( int c );
```

Returns a non–zero value when **c** is a punctuation character (such as '.', ',', '!', etc.).

isspace

```
#include <ctype.h>
_regparm int isspace( int c );
```

Returns a non–zero value when **c** is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).

isupper

```
#include <ctype.h>
_regparm int isupper( int c );
```

Returns a non–zero value when **c** is an uppercase character ([A–Z]).

isxdigit

```
#include <ctype.h>
_regparm int isxdigit( int c );
```

Returns a non–zero value when `c` is a hexadecimal digit ([0–9][A–F][a–f]).

labs

```
#include <stdlib.h>
_regparm long labs( long n );
```

Returns the absolute value of the signed long argument.

ldexp

```
#include <math.h>
_regparm double ldexp( double x, int n );
```

Returns the result of: $x \cdot 2^n$.

ldiv

```
#include <stdlib.h>
_regparm ldiv_t ldiv( long num, long denom );
```

Both arguments are long integers. The returned quotient and remainder are also long integers.

Returns a structure containing the quotient and remainder of `num` divided by `denom`.

log

```
#include <math.h>
_regparm double log( double x );
```

Returns the natural logarithm `ln(x), x>0`.

log10

```
#include <math.h>
_regparm double log10( double x );
```

Returns the base 10 logarithm `log10(x), x>0`.

longjmp

```
#include <setjmp.h>
_regparm void longjmp( jmp_buf env, int val );
```

Restores the environment previously saved with a call to setjmp(). The function calling the corresponding call to setjmp() may not be terminated yet. The value of `val` should not be zero.

Returns nothing.

malloc

```
#include <stdlib.h>
_regparm void xdat *malloc( size_t size );
```

The allocated space is not initialized. The maximum space that can be allocated can be changed by customizing the heap size (see section 7.5, *Heap*). By default no heap is allocated. When ”malloc()” is used while no heap is defined, the linker gives an error.

Be aware, for ’small’ and ’aux’ model, pointers to the allocated area must be declared as pointing to `xdat`.

Returns a pointer to space in external memory of `size` bytes length. NULL if there is not enough space left.

memchr

```
#include <string.h>
_regparm void *memchr( const void *cs, int c,
                       size_t n );
```

Checks the first `n` bytes of `cs` on the occurrence of character `c`.

Returns NULL when not found, otherwise a pointer to the found character is returned.

memcmp

```
#include <string.h>
_regparm int memcmp( const void *cs,
                     const void *ct, size_t n );
```

Compares the first `n` bytes of `cs` with the contents of `ct`.

Returns a value < 0 if `cs < ct`,
0 if `cs = = ct`,
or a value > 0 if `cs > ct`.

memcpy

```
#include <string.h>
_regparm void *memcpy( void *s,
                       const void *ct, size_t n );
```

Copies `n` characters from `ct` to `s`. No care is taken if the two objects overlap.

Returns `s`

memmove

```
#include <string.h>
_regparm void *memmove( void *s,
                        const void *ct, size_t n );
```

Copies `n` characters from `ct` to `s`. Overlapping objects will be handled correctly.

Returns `s`

memset

```
#include <string.h>
_regparm void *memset( void *s, int c,
                       size_t n );
```

Fills the first `n` bytes of `s` with character `c`.

Returns `s`

modf

```
#include <math.h>
_regparm double modf( double x, double *ip );
```

Splits `x` into integral and fractional parts, each with the same sign as `x`. It stores the integral part in `*ip`.

Returns the fractional part.

offsetof

```
#include <stddef.h>
_regparm int offsetof( type, member );
```

Returns the offset for the given `member` in an object of type.

Be aware, offsetof() for bitstructures/unions may give unpredictable results. Also the offsetof() of a bit field is undefined.

pow

```
#include <math.h>
_regparm double pow( double x, double y );
```

A domain error occurs if `x=0` and `y<=0`, or if `x<0` and `y` is not an integer.

Returns the result of `x` raised to the power of `y`: x^y.

printf

```
#include <stdio.h>
_regparm int printf( const char *format, ... );
```

Performs a formatted write to the standard output stream.

See also "_iowrite()" and section 6.3.3, *Printf and Scanf Formatting Routines*.

Returns the number of characters written to the output stream.

The `format` string may contain plain text mixed with conversion specifiers. Each conversion specifier should be preceded by a '%' character. The conversion specifier should be build in order:

- Flags (in any order):
  - `-` specifies left adjustment of the converted argument.
  - `+` a number is always preceded with a sign character. `+` has higher precedence as space.
  - `space` a negative number is preceded with a sign, positive numbers with a space.
  - `0` specifies padding to the field width with zeros (only for numbers).
  - `#` specifies an alternate output form. For o, the first digit will be zero. For x or X, "0x" and "0X" will be prefixed to the number. For e, E, f, g, G, the output always contains a decimal point, trailing zeros are not removed.

- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag '−' was specified) with spaces. Padding to numeric fields will be done with zeros when the flag '0' is also specified (only when padding left). Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type int.
- A period. This separates the minimum field width from the precision.
- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type int.
- A length modifier 'h', 'l' or 'L'. 'h' indicates that the argument is to be treated as a short or unsigned short number. 'l' should be used if the argument is a long integer. 'L' indicates that the argument is a long double.

Flags, length specifier, period, precision and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined:

Character	Printed as
d, i	int, signed decimal
o	int, unsigned octal
x, X	int, unsigned hexadecimal in lowercase or uppercase respectively
u	int, unsigned decimal
c	int, single character (converted to unsigned char)
s	char *, the characters from the string are printed until a NULL character is found. When the given precision is met before, printing will also stop
f	double
e, E	double
g, G	double

Character	Printed as
n	int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed.
%	No argument is converted, a '%' is printed.

Table 6–3: Printf conversion characters

The 'p' conversion character is not supported because the formatter is unable to know to which memory the pointer is pointing to. However, you can print pointers as unsigned decimal ('u' conversion character).

putc

```
#include <stdio.h>
_regparm int putc( int c, FILE *stream );
```

Puts one character onto the given stream.

See also "_iowrite()".

Returns EOF on error.

putchar

```
#include <stdio.h>
_regparm int putchar( int c );
```

Puts one character onto standard output.

See also "_iowrite()".

Returns the character written or EOF on error.

puts

```
#include <stdio.h>
_regparm int puts( const char *s );
```

Writes the string to stdout, the string is terminated by a newline.

See also "_iowrite()".

Returns NULL if successful, or EOF on error.

qsort

```
#include <stdlib.h>
_regparm _reentrant void qsort( void *base,
  size_t n, size_t size,
  _regparm _reentrant int (* cmp)(const void *,
  const void *) );
```

This function sorts an array of `n` members. The initial base of the array is given by `base`. The size of each member is specified by `size`. The given array is sorted in ascending order, according to the results of the function pointed to by `cmp`.

rand

```
#include <stdlib.h>
_regparm int rand( void );
```

Returns a sequence of pseudo–random integers, in the range 0 to RAND_MAX.

realloc

```
#include <stdlib.h>
_regparm void xdat *realloc( void *p,
                             size_t size );
```

Reallocates the space for the object pointed to by `p`. The contents of the object will be the same as before calling realloc().The maximum space that can be allocated can be changed by customizing the heap size (see section 7.5, *Heap*). By default no heap is allocated. When ”realloc()” is used while no heap is defined, the linker gives an error.

See also ”malloc()”.

Returns NULL and `*p` is not changed, if there is not enough space for the new allocation. Otherwise a pointer to the newly allocated space for the object is returned.

romidcpy

```
#include <string.h>
_regparm void idat *romidcpy( idat void *cs,
                              const rom void *ct,
                              size_t n );
```

Copies `n` bytes of data from ROM memory to `idat` memory.

See also ”ididcpy()”.

romidmove

```
#include <string.h>
_regparm void idat *romidmove( idat void *cs,
                               const rom void *ct,
                               size_t n );
```

Moves `n` bytes of data from ROM memory to `idat` memory.

See also ”ididcpy()”.

romxdcpy

```
#include <string.h>
_regparm void xdat *romxdcpy( xdat void *cs,
                              const rom void *ct,
                              size_t n );
```

Copies `n` bytes of data from ROM memory to `xdat` memory.

See also ”ididcpy()”.

romxdmove

```
#include <string.h>
_regparm void xdat *romxdmove( xdat void *cs,
                               const rom void *ct,
                               size_t n );
```

Moves `n` bytes of data from ROM memory to `xdat` memory.

See also "ididcpy()".

scanf

```
#include <stdio.h>
_regparm int scanf( const char *format, ... );
```

Performs a formatted read from the standard input stream.

See also "_ioread()" and section 6.3.3, *Printf and Scanf Formatting Routines*.

Returns the number of items converted successfully.

All arguments to this function should be pointers to variables (in default memory) of the type which is specified in the format string.

The format string may contain :

- Blanks or tabs, which are skipped.
- Normal characters (not '%'), which should be matched exactly in the input stream.
- Conversion specifications, starting with a '%' character.

Conversion specifications should be build as follows (in order) :

- A '*', meaning that no assignment is done for this field.
- A number specifying the maximum field width.
- A length modifier 'h', 'l' or 'L'. 'h' indicates the argument is to be treated as a short or unsigned short number. 'l' should be used if the argument is a long integer. 'L' indicates that the argument is a long double.

– A conversion specifier. '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Character	Scanned as
d	int, signed decimal.
i	int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading "0x" or "0X"), or just decimal.
o	int, unsigned octal.
u	int, unsigned decimal.
x	int, unsigned hexadecimal in lowercase or uppercase.
c	single character (converted to unsigned char).
s	char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.
f	float
e, E	float
g, G	float
n	int *, the number of characters written so far is written into the argument. No scanning is done.
[...]	Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying []...] includes the ']' character in the set of scanning characters.
[^...]	Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^]...] includes the ']' character in the set.
%	Literal '%', no assignment is done.

Table 6–4: Scanf conversion characters

The 'p' specifier is not supported because the formatter is unable to know to which memory the pointer is pointing to.

setjmp

```
#include <setjmp.h>
_regparm int setjmp( jmp_buf env );
```

Saves the current environment for a subsequent call to longjmp.

Returns the value 0 after a direct call to setjmp(). Calling the function "longjmp()" using the saved `env` will restore the current environment and jump to this place with a non–zero return value.

See also "longjmp()".

sin

```
#include <math.h>
_regparm double sin( double x );
```

Returns the sine of `x`.

sinh

```
#include <math.h>
_regparm double sinh( double x );
```

Returns the hyperbolic sine of `x`.

sprintf

```
#include <stdio.h>
_regparm int sprintf( char *s,
                      const char *format, ... );
```

Performs a formatted write to a string.

See also "printf()" and section 6.3.3, *Printf and Scanf Formatting Routines*.

sqrt

```
#include <math.h>
_regparm double sqrt( double x );
```

Returns the square root of `x`. $\sqrt{x}$, where $x \geq 0$.

srand

```
#include <stdlib.h>
_regparm void srand( unsigned int seed );
```

This function uses `seed` as the start of a new sequence of pseudo–random numbers to be returned by subsequent calls to srand(). When srand is called with the same seed value, the sequence of pseudo–random numbers generated by rand() will be repeated.

Returns pseudo random numbers.

sscanf

```
#include <stdio.h>
_regparm int sscanf( char *s,
                     const char *format, ... );
```

Performs a formatted read from a string.

See also ”scanf()” and section 6.3.3, *Printf and Scanf Formatting Routines*.

strcat

```
#include <string.h>
_regparm char *strcat( char *s,
                       const char *ct );
```

Concatenates string ct to string `s`, including the trailing NULL character.

Returns `s`

strchr

```
#include <string.h>
_regparm char *strchr( const char *cs, int c );
```

Returns a pointer to the first occurrence of character `c` in the string `cs`. If not found, NULL is returned.

strcmp

```
#include <string.h>
_regparm int strcmp( const char *cs,
                     const char *ct );
```

Compares string `cs` to string `ct`.

Returns <0 if `cs < ct`,
0 if `cs == ct`,
>0 if `cs > ct`.

strcpy

```
#include <string.h>
_regparm char *strcpy( char *s,
                       const char *ct );
```

Copies string `ct` into the string `s`, including the trailing NULL character.

Returns `s`

strcspn

```
#include <string.h>
_regparm size_t strcspn( const char *cs,
                         const char *ct );
```

Returns the length of the prefix in string `cs`, consisting of characters not in the string `ct`.

strlen

```
#include <string.h>
_regparm size_t strlen( const char *cs );
```

Returns the length of the string in `cs`, not counting the NULL character.

strncat

```
#include <string.h>
_regparm char *strncat( char *s,
                        const char *ct, size_t n );
```

Concatenates string `ct` to string `s`, at most `n` characters are copied. Add a trailing NULL character.

Returns `s`

strncmp

```
#include <string.h>
_regparm int strncmp( const char *cs,
                      const char *ct, size_t n );
```

Compares at most `n` bytes of string `cs` to string `ct`.

Returns <0 if `cs < ct`,
0 if `cs == ct`,
>0 if `cs > ct`.

strncpy

```
#include <string.h>
_regparm char *strncpy( char *s,
                          const char *ct, size_t n );
```

Copies string `ct` onto the string `s`, at most `n` characters are copied. Add a trailing NULL character if the string is smaller than `n` characters.

Returns `s`

strpbrk

```
#include <string.h>
_regparm char *strpbrk( const char *cs,
                        const char *ct );
```

Returns a pointer to the first occurrence in `cs` of any character out of string `ct`. If none are found, NULL is returned.

strrchr

```
#include <string.h>
_regparm char *strrchr( const char *cs,
                        int c );
```

Returns a pointer to the last occurrence of `c` in the string `cs`. If not found, NULL is returned.

strspn

```
#include <string.h>
_regparm size_t strspn( const char *cs,
                        const char *ct );
```

Returns the length of the prefix in string `cs`, consisting of characters in the string `ct`.

strstr

```
#include <string.h>
_regparm char *strstr( const char *cs,
                       const char *ct );
```

Returns a pointer to the first occurrence of string `ct` in the string `cs`. Returns NULL if not found.

strtod

```
#include <stdlib.h>
_regparm double strtod( const char *s,
                        char **endp );
```

Converts the initial portion of the string pointed to by `s` to a double value. Initial white spaces are skipped. When `endp` is not a NULL pointer, after this function is called, `*endp` will point to the first character not used by the conversion.

Returns the read value.

strtok

```
#include <string.h>
_regparm char *strtok( char *s,
                       const char *ct );
```

Search the string `s` for tokens delimited by characters from string `ct`. It terminates the token with a NULL character.

Returns a pointer to the token. A subsequent call with `s` == NULL will return the next token in the string.

strtol

```
#include <stdlib.h>
_regparm long strtol( const char *s,
                      char **endp, int base );
```

Converts the initial portion of the string pointed to by `s` to a long integer. Initial white spaces are skipped. Then a value is read using the given `base`. When `base` is zero, the `base` is taken as defined for integer constants. I.e. numbers starting with an '0' are taken octal, numbers starting with '0x' or '0X' are taken hexadecimal. Other numbers are taken decimal. When `endp` is not a NULL pointer, after this function is called, `*endp` will point to the first character not used by the conversion.

Returns the read value.

strtoul

```
#include <stdlib.h>
_regparm unsigned long strtoul( const char *s,
                                char **endp, int base );
```

Converts the initial portion of the string pointed to by `s` to an unsigned long integer. Initial white spaces are skipped. Then a value is read using the given `base`. When `base` is zero, the `base` is taken as defined for integer constants. I.e. numbers starting with an '0' are taken octal, numbers starting with '0x' or '0X' are taken hexadecimal. Other numbers are taken decimal. When `endp` is not a NULL pointer, after this function is called, `*endp` will point to the first character not used by the conversion.

Returns the read value.

tan

```
#include <math.h>
_regparm double tan( double x);
```

Returns the tangent of `x`.

tanh

```
#include <math.h>
_regparm double tanh( double x);
```

Returns the hyperbolic tangent of `x`.

time

```
#include <time.h>
time_t time( time_t *tp );
```

The return value is also assigned to `*tp`, if `tp` is not NULL.

Returns the current calendar time, or –1 if the time is not available.

toascii

```
#include <ctype.h>
_regparm int toascii( int c );
```

Converts `c` to an ascii value (strip highest bit). This is a non–ANSI function.

Returns the converted value.

tolower

```
#include <ctype.h>
_regparm int tolower( int c );
```

Returns `c` converted to a lowercase character if it is an uppercase character, otherwise `c` is returned.

toupper

```
#include <ctype.h>
_regparm int toupper( int c );
```

Returns c converted to an uppercase character if it is a lowercase character, otherwise c is returned.

ungetc

```
#include <stdio.h>
_regparm int ungetc( int c, FILE *fin );
```

Pushes at the most one character back onto the input buffer.

Returns EOF on error.

va_arg

```
#include <stdarg.h>
_regparm va_arg( va_list ap, type );
```

Returns the value of the next argument in the variable argument list. It's return type has the type of the given argument `type`. A next call to this macro will return the value of the next argument.

va_end

```
#include <stdarg.h>
_regparm va_end( va_list ap );
```

This macro must be called after the arguments have been processed. It should be called before the function using the macro 'va_start' is terminated (ANSI specification).

va_start

```
#include <stdarg.h>
_regparm va_start( va_list ap, lastarg );
```

This macro initializes `ap`. After this call, each call to va_arg() will return the value of the next argument. In our implementation, `va_list` cannot contain any bit type variables. Also the given argument `lastarg` must be the last non–bit type argument in the list.

vfprintf

```
#include <stdio.h>
_regparm int vfprintf( FILE *stream,
               const char *format, va_list arg );
```

Is equivalent to vprintf, but writes to the given stream.

See also "vprintf()", "_iowrite()" and section 6.3.3, *Printf and Scanf Formatting Routines*.

vprintf

```
#include <stdio.h>
_regparm int vprintf( const char *format,
                      va_list arg );
```

Does a formatted write to standard output. Instead of a variable argument list as for printf(), this function expects a pointer to the list.

See also ”printf()”, ”_iowrite()” and section 6.3.3, *Printf and Scanf Formatting Routines*.

vsprintf

```
#include <stdio.h>
_regparm int vsprintf( char *s,
                  const char *format, va_list arg );
```

Does a formatted write a string. Instead of a variable argument list as for printf(), this function expects a pointer to the list.

See also ”printf()”, ”_iowrite()” and section 6.3.3, *Printf and Scanf Formatting Routines*.

xdidcpy

```
#include <string.h>
_regparm void idat *xdidcpy( idat void *cs,
                             const xdat void *ct,
                             size_t n );
```

Copies `n` bytes of data from `xdat` memory to `idat` memory.

See also ”ididcpy()”.

xdidmove

```
#include <string.h>
_regparm void idat *xdidmove( idat void *cs,
                              const xdat void *ct,
                              size_t n );
```

Moves `n` bytes of data from `xdat` memory to `idat` memory.

See also "ididcpy()".

xdxdcpy

```
#include <string.h>
_regparm void xdat *xdxdcpy( xdat void *cs,
                             xdat void *ct,
                             size_t n );
```

Copies `n` bytes of data from `xdat` memory to `xdat` memory.

See also "ididcpy()".

xdxdmove

```
#include <string.h>
_regparm void xdat *xdxdmove( xdat void *cs,
                              xdat void *ct,
                              size_t n );
```

Moves `n` bytes of data from `xdat` memory to `xdat` memory. Overlapping spaces are handled correctly.

See also "ididcpy()".

6.3.3 PRINTF AND SCANF FORMATTING ROUTINES

The functions `printf()`, `fprintf()`, `vfprintf()`, `vsprintf()`, ... call one single function that deals with the format string and arguments. This function is `_doprint()`. This is a rather big function because the number of possibilities of the format specifiers in a format string are large. If you do not use all the possibilities of the format specifiers a smaller `_doprint()` function can be used. Three different versions exist:

LARGE	the full formatter, no restrictions
MEDIUM	floating point printing is not supported
SMALL	as MEDIUM, but also the precision specifier '.' cannot be used

The same applies to all `scanf` type functions, which all call the function `_doscan()`. Two different versions exist:

MEDIUM	the full formatter, no restrictions
SMALL	floating point printing is not supported

Special versions of the formatters are installed which can handle a format string placed in ROM.

The default printf formatter used in the C library is the MEDIUM version (`printfsm.lib`). The default scanf formatter used in the C library is the SMALL version (`scanfss.lib`). You can select different formatters by linking separate libraries with your application.

The printf/scanf libraries included with the product have the following name syntax:

printf{*model*}{*pversion*}[**f**]**.lib**
scanf{*model*}{*sversion*}[**f**]**.lib**

where,

model	indicates the compiler model **s** (small), **a** (auxpage), **l** (large) or **r** (reentrant)
pversion	indicates the printf formatter version **s** (small), **m** (medium) or **l** (large)
sversion	indicates the scanf formatter version **s** (small) or **m** (medium)
f	(optionally) indicates that the format string is in ROM

When you use EDE, you can specify the printf/scanf libraries in the `Compiler | Printf/Scanf` entry of the `Project | Project Options...` dialog.

Section 3.4, *Function Parameters* and section 3.10, *Strings*.

6.4 RUN-TIME LIBRARY

Some compiler generated code contains calls to run–time library functions that would use too much code if generated as inline code. The name of a run–time library function always contains two leading underscores. For example, to perform a 16 bit unsigned division on two 'register–pairs', the function __UDIVI is called.

Because **cc51** generates assembly code (and not object code) it prepends an underscore for the names of (public) C variables to distinguish these symbols from 8051 registers. So if you use a function with a leading underscore, the assembly label for this function contains two leading underscores. This function name could cause a name conflict (double defined) with one of the run–time library functions. However, ANSI states that it is not portable to use names starting with an underscore for public C variables and functions, because results are implementation defined.

All code of the run–time library functions is placed in a CODE segment called ?C51RTL_PR.

The run–time library functions are included in the C library.

6.5 CREATING YOUR OWN C LIBRARY

There are several reasons why it is desired to have a specially adapted C library. Therefore all C sources of all library functions are delivered with the compiler (this file is placed in the directory `cc51\lib\src` when using PC, `/usr/local/cc51/lib/src` when using UNIX).

When creating your own library, the order of the objects in the library file is very important. To know the exact order in which the objects should be placed in the library, make a list of the order in which the delivered libraries are made by using the command '**ar51 t c51r.lib**' (or **c51l.lib** or one of the other libraries).

The easiest method to create your own library is to make a copy of the existing library (use the library in the same memory model you want to create) and replace the existing objects in it by your own made objects with the command '**ar51 crv** *libname objectname* ...' . This way the order of the objects in the library will be maintained. At link time you only have to link the newly made library to your application instead of a delivered library.

When starting with an empty library you have to link the original library also, because it contains all run–time routines needed to run your application. In this case the original library must be the last file specified to **link51**.

CHAPTER 7

RUN-TIME ENVIRONMENT

TASKING

1 CHAPTER

7.1 STARTUP CODE

When linking your C modules with the library, you must also link the object module, containing the C startup code. This file is called `cstart.obj`.

Because this module specifies the run–time environment of your C application, you might want to edit it to match your needs. Therefore, this module is delivered in assembly source in the file `cstart.asm` in the `lib/src` subdirectory. Typically, you will copy the template startup file to your own directory and edit it. The startup code contains preprocessor symbols that can be interpreted by **mpp51** when you want to make your own version of the object file.

EDE is capable of generating the startup code automatically. To do this: open the `Project | Project Options` dialog, expand the `Processor` entry and select `Startup Code`. Enable the options `Generate startup code (<project>_cstart.asm)` and `Add startup code (<project>_cstart.asm) to your project`.

Table 7–1 shows all the macros (defines) that can be set for `cstart.asm`. The defines can be set using **mpp51** command line option **–D***macro*=*value* or within EDE in the `Processor | Startup Code` and `Assembler | Macro Preprocessor` pages of the `Project | Project Options` dialog.

Define	Default	Description
CLR_EA		setting this define makes sure all interrupts are disabled at startup; by default the startup code will not clear the interrupt enable all bit (EA)
FLOATMEM	0	define the number of floats on the floating point stack
HEAP	0	define the heap size
MFLOAT	XDATA	define to IDATA when the floating point stack is in IDAT instead of XDAT (link floats.lib in that case instead of float.lib).
MODEL	SMALL	define the memory model (SMALL, AUX, LARGE or REENTRANT)
MON51	NO	define to TASKING when using the TASKING Rom monitor define to RISM51 when using the Intel Rism51 monitor.

Define	Default	Description
P2	0	set port P2 pins at startup; this defines the external RAM page to be used for paged data (_pdat)
PROTECT	NO	define YES only for reentrant model with interrupt functions using the virtual stack
RAMSIZE	080H	size of internal (IDAT) memory (128–256 bytes) to be cleared at startup
REGBANK	0	define the default register bank at startup
STACKLENGTH	20H	define a (minimum) stack length
SYSCON		certain Infineon Technologies derivates have a SYSCON SFR which can be used to direct 8–bit MOVX instructions to internal XRAM. Defining this macro to the correct value will result in correct initialization of this SFR at startup
VSTACK	NO	define YES if at least one function is declared _reentrant while the memory model is not reentrant
VIRT_STACK	400H	define the virtual stack size (only for reentrant model)
XDATSTART XDATEND	0 0	specify the start and end of the XDAT area to be cleared at startup

Table 7–1: Macros used in `cstart.asm`

The startup code contains macro preprocessor symbols, so you must use **mpp51** before **asm51** when you want to make a new version of the object file:

```
mpp51 cstart.asm
asm51 cstart noprint
```

In the C startup code an absolute code segment is defined for setting up the power on vector and the C–51 environment. The power–on vector contains a jump to the __START label, which is placed after all other interrupt vectors. The code space for all non used interrupt vectors may be occupied by small user code segments. When this is not wanted, you should allocate the space for all non used interrupt vectors in the startup code. Thus preventing **link51** from using this area for a user code segment. If you are using interrupts, you should not allocate the space at the addresses of the interrupt vector, because the real interrupt vectors are loaded from the library. If you do allocate this space, **link51** will warn you with the message: "CODE SPACE MEMORY OVERLAP".

The stack is defined in a segment called ?STACK, because **link51** allocates this segment after all other IDATA segments. The public symbol __STKSTART must be present, because it is used by both a debugger and the library function `exit()`. The stack size can be controlled with the macro preprocessor symbol STACKLENGTH, which defaults to 32 bytes. Remember that there must be enough space allocated in this ?STACK segment for the stack, which grows upwards.

When using the reentrant model or when some functions are programmed `_reentrant`, a virtual stack is needed. The size of this stack (which is placed in external RAM) is defined by the preprocessor symbol VIRT_STACK. When not using the reentrant model, but some functions are programmed `_reentrant`, allocation of the virtual stack and initialization of the virtual stack pointer is forced by defining the preprocessor symbol VSTACK.

When using the reentrant model or when functions are programmed `_reentrant`, it may be needed to change the value of PROTECT in the `cstart.asm` file. In that case, please read section 7.10, *Reentrant Model / _reentrant Functions*.

The heap size is also defined using a macro preprocessor symbol. The heap area is allocated in XDATA.

See section 7.5, *Heap*, for detailed information on heap management.

When using floating point in your application, you should define the size of a floating point stack.

See also section 7.6, *Floating Point*.

All available internal RAM and external RAM must be cleared, because in C all non initialized static variables are defined to have a value of 0 at startup. This is done by the C startup code. The internal RAM size is defined with the macro preprocessor symbol RAMSIZE, which normally contains a value between 128 (e.g. 8051) and 256 (e.g. 8052, 80C552). The start address and end address of external RAM are defined with the preprocessor symbols XDATSTART and XDATEND. By default, no memory will be cleared by the startup code. When it is needed that memory is cleared at startup, this should be changed in the startup code.

The startup code also takes care of initialized C variables, residing in the different RAM areas. Each memory type has a unique name for both the ROM and the RAM segment. The startup code copies the initial values of initialized C variables from ROM to RAM, using these special segments and some run–time library functions. A special segment is used for strings in ROM, which are copied to the appropriate RAM segment, depending on the memory model used for the C modules. Therefore you must specify the memory model you are using to **mpp51** before it processes the startup code. This can be done by defining the preprocessor symbol MODEL. The startup file uses the define MODEL to select the RAM area used as destination for the strings.

When everything described above has been executed, your C application is called, using the public label `_?main`, which has been generated by **cc51** for the C function `main()`.

When the C application 'returns', which is not likely to happen in an embedded environment, the program performs an endless loop, using the assembly label __STOP. When using a debugger, it can be useful to set a breakpoint on this label, indicating the program has reached the end, or the library function `exit()` has been called.

7.2 REGISTER USAGE

In all models **cc51** uses the following 8051 registers for code generation: R0–R7, A, B, DPTR and PSW. When calling a user assembly routine from C, none of these registers need to be saved by the assembly routine, because these registers are used for temporary results only. When one of these registers has a temporary result, the compiler saves it on stack before the assembly language routine is called, and restores it afterwards.

cc51 uses the following registers for C function return types:

Return type	Register	Description
bit	C	(carry)
char	A	(accumulator)
short/int	R6–R7	(R6 high byte, R7 low byte)
long	R4–R7	(R45 high word, R67 low word)
float	–	(Floating point stack)
near pointer	A	(accumulator)
far pointer	R6–R7	(R6 high byte, R7 low byte)

Table 7–2: Register usage

7.3 SEGMENT USAGE

cc51 uses a large number of segments. This section contains a list of all possible segment names of a complete C application:

BIT

C51_BI	user C bit type variables
CINIT_RAM_BI	initialized C bit variables
BSEG AT xx	absolute segments for variables placed using the `_at()` keyword
FP_BIT	floating point data used by run–time routines

DATA

C51_DA	user C variables residing in `data`
CINIT_RAM_DA	initialized C variables residing in `data`
C51_BA	user C variables residing in bitaddressable `data`
CINIT_RAM_BA	initialized C variables residing in bitaddressable `data`
DSEG AT xx	absolute segments for variables placed using the `_at()` keyword
DRSEG	floating point data used by run–time routines
DBRSEG	bitaddressable floating point data used by run–time routines

IDATA

C51_I	user C variables residing in `idat`
CINIT_RAM_DI	initialized C variables residing in `idat`
?STACK	last IDATA segment, for stack allocation
CINIT_RAM_ST	for small model only, string area
ISEG AT xx	absolute segments for variables placed using the `_at()` keyword

CODE

CSEG AT 0H	absolute code segment for power on vector and startup code.
STARTUP	C Startup Code
C51_PR	user C functions
C51LIB_PR	C library functions
?C51RTL_PR	run–time library functions
LIB_FP	floating point run–time routines
CSEG AT xx	absolute code segments for interrupt vectors
C51_CO	user C variables residing in `rom`, switch tables and 'romstrings'
CINIT_ROM_BI	initial values of user initialized C bit variables
CINIT_ROM_DA	initial values of user initialized C variables residing in `data`
CINIT_ROM_BA	initial values of user initialized C variables residing in bitaddressable `data`
CINIT_ROM_ID	initial values of user initialized C variables residing in `idat`
CINIT_ROM_PD	initial values of user initialized C variables residing in `pdat`
CINIT_ROM_XD	initial values of user initialized C variables residing in `xdat`
CINIT_ROM_ST	strings (residing in either `idat`, `pdat` or `xdat`, depending on memory model used)

XDATA

C51_PD	user C variables residing in `pdat`
CINIT_RAM_PD	initialized C variables residing in `pdat`
C51_XD	user C variables residing in `xdat`
CINIT_RAM_XD	initialized C variables residing in `xdat`

?HEAP	allocation of heap area
CINIT_RAM_ST	for all models but small, string area
XSEG AT xx	absolute segments for variables placed using the `_at()` keyword
?VIRT_STACK	virtual stack space used by `_reentrant` functions
FP_XDAT	floating point data used by run–time routines

If overlaying is used (default for non–reentrant functions), more segments are declared containing the module name, function name, memory type and register bank involved.

The segment CINIT_RAM_ST (RAM area for strings) is allocated in either `idat`, `pdat` or `xdat`, depending on the memory model (MODEL) used in the C startup code. Default is `idat`. For details on changing startup code, see section 7.1, *Startup Code*.

If you use the **–R** option, to specify the name **cc51** must use for a certain segment, this name is added to this list. Note that **link51** produces a link map (suffix `.l51`) which shows the addresses of all segments used in the application.

Segment renaming is only possible in the reentrant memory model. In other memory models the segment names need to be fixed in order for the overlaying mechanism to work correctly.

7.4 STACK

The following diagrams show the structure of the stack. The first diagram reflects the system stack. The second diagram shows the virtual stack when using reentrant functions.

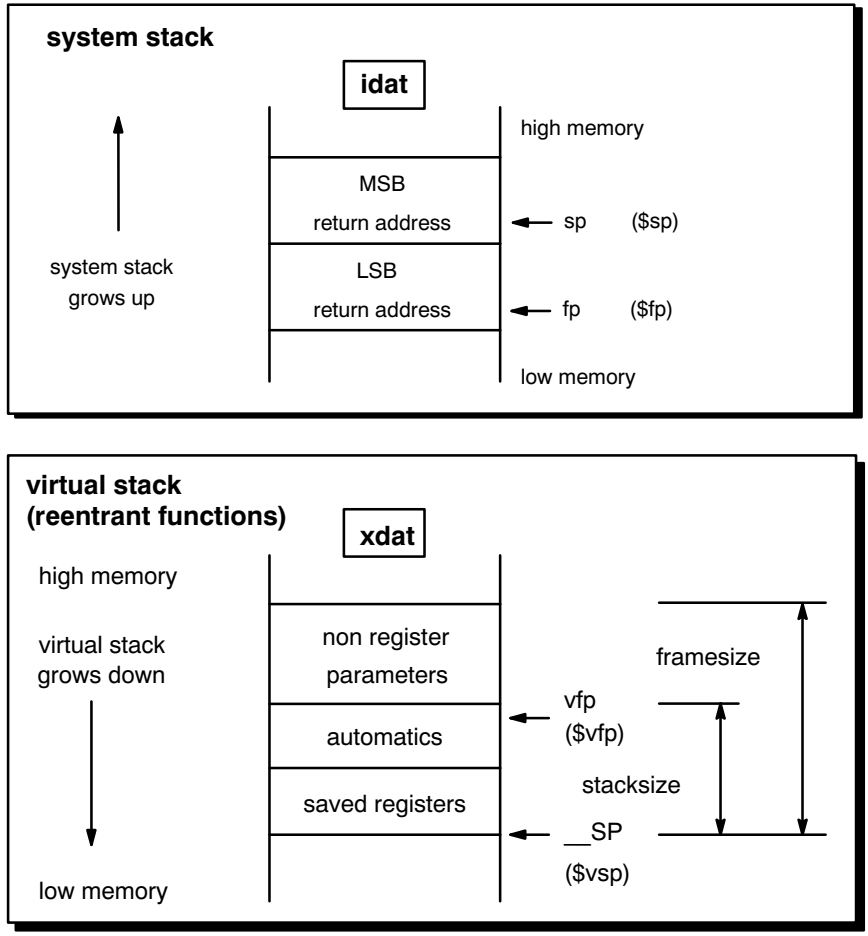


Figure 7–1: Stack diagrams

The **system stack** is used (using direct internal RAM) for return addresses only. The stack is allocated via the ?STACK segment. You can specify the size of the stack segment in the C startup code (`cstart.asm`). **link51** locates the ?STACK segment as the last indirect addressable internal RAM segment (after all the user IDATA segments), because the system stack is growing from low to high. For `_small`, `_aux` and, `_large` functions, automatics and parameters are allocated in overlayable data sections, and therefore, do not use any stack space.

In EDE you can enter the system stack size in the `System stack size` field in the `Linker | Stack/Heap` entry of the `Project | Project Options...` dialog.

The label `__STKSTART` (also present in `cstart.asm`) is used as the absolute bottom of the system stack. If this label is not present, the system reset value of the SP register is used (at address `data:0x7`).

For `_reentrant` functions, a **virtual stack** is used in external RAM. Automatics and parameters are all accessed using a virtual stack pointer register, allocated as a 16–bit pointer in direct addressable internal RAM (label `__SP`). The stack frame also contains a so–called virtual frame pointer, which can be seen as a frame pointer register for debugging purposes, and therefore, is supported by CrossView Pro as a pseudo register called `$vfp`. The saved registers are also accessed using a virtual stack pointer. The virtual stack pointer can be seen as a virtual stack pointer register, and therefore, is supported by CrossView Pro as a pseudo register called `$vsp`.

In EDE you can enter the virtual stack size: enable the option `Application uses reentrant functions` in the `Linker | Stack/Heap` entry of the `Project | Project Options...` dialog and enter a size in the `Virtual stack size` field.

Run time routines are called for a function's prologue, epilogue and automatic/parameter access.

The label `__TOP_OF_VIRT_STACK` (`cstart.asm`) is used as the absolute top of the virtual stack. When using `_reentrant` functions, this label, and of course `__SP`, should be present.

7.5 HEAP

The heap is only needed when dynamic memory management library functions are used: `malloc()`, `calloc()`, `free()` and `realloc()`. The heap is a reserved area in external RAM with a default size of 0 bytes. If you use one of the memory allocation functions listed above, the linker will give errors if no heap is defined. So when you want to use one of these routines, you must change the heap size in the startup code.

The macro preprocessor symbol HEAP is used to define the size of the heap. A special XDAT segment called ?HEAP is used for the allocation of the heap area. You can place the heap segment anywhere in memory, using a linker command file specifying either the order of allocation or an absolute address. The public assembly symbols __HEAPSTART and __HEAPLENGTH are used by the library function `sbrk()`, which is called by `malloc()` when memory is needed from the heap.

After editing, you must process the C startup file with both **mpp51** and **asm51** to make the correct object file. For a detailed description, see section 7.1, *Startup Code*.

7.6 FLOATING POINT

cc51 has implemented single precision floating point arithmetic, i.e. 'double' and 'long double' variables are treated as normal 'float' variables.

Floating point operators use a special floating point stack area which should be defined within the startup code. This area is placed in external RAM and has a default size of 0 bytes. You must change the size (in the startup code) to be able to use floating point.

A special floating point library is delivered to support floating point arithmetic when no external RAM is available. This library is called `floats.lib`. This library uses a floating point stack in internal RAM (`idat` space). When you use this library, no math functions are available within the library. In the startup code you have to specify that the floating point stack is located in internal RAM (see the startup file `cstart.asm` for more information). If you do not change the startup code, the linker will produce error messages. Placing the floating point stack in internal RAM does not significantly increase floating point arithmetic.

When your application uses floating point arithmetic, be aware of the following:

- Define a floating point stack in the startup code, all operations and temporary results are placed on this stack. For very complex expressions, the stack must be large enough to hold all temporary results, but normally 5 elements will do.
- Floating point is not reentrant. No floating point arithmetic or even assignments can be done on interrupt.
- The floating point library `float.lib` must be specified to the linker as the last library in the list. Also the C library must be linked before `float.lib`.
- Due to the very limited internal RAM of a 80C751 derivative (only 64 bytes), floating point is not supported.

7.7 INTERRUPT FUNCTIONS

Interrupt functions may be implemented directly in C, by using the `_interrupt(`*n*`)` or `__interrupt(`*addr*`)` function qualifier. A function declared with this qualifier differs from a normal function definition in a number of ways:

1. The appropriate interrupt vector, consisting of a JMP instruction jumping to the interrupt function is generated. The vector may be suppressed with `_interrupt(-1)` with the **–v** option or the **#pragma novector**.

2. All non R0–R7 registers A, B, DPTR and PSW that might possibly be corrupted during the execution of the interrupt function are saved on function entry and restored on function exit. The compiler will check the function to see which of these registers are being used and automatically save/restore only those registers. When the `_using()` qualifier is used the registers R0–R7 are implicitly saved when the register bank is being switched (by using the predefined symbolic register addresses AR0–AR7). When this qualifier is not used the compiler will check the function and save/restore only those registers.

3. The function is terminated with a RETI instruction instead of a RET instruction.

Example:

```
; 8051 C compiler vx.y rz         SNaaaaaa (c) year TASKING, Inc.
; options: -s
$CASE
        NAME    INTRPT
; intrpt.c    1 int x,y;
        PUBLIC  _x
C51_DA  SEGMENT DATA
        RSEG    C51_DA
_x:     DS      2
        PUBLIC  _y
_y:     DS      2
; intrpt.c    2
; intrpt.c    3 _interrupt(17) void int17( void )
; intrpt.c    4 {
        PUBLIC  _?int17
        CSEG AT 08BH
        LJMP    _?int17
; free registers in this function: B DPTR R1 R2 R3
INTRPT_INT17_PR SEGMENT CODE
        RSEG    INTRPT_INT17_PR
_?int17:
        USING   0
        PUSH    ACC
```

```
        PUSH    AR0
        PUSH    AR4
        PUSH    AR5
        PUSH    AR6
        PUSH    AR7
        PUSH    PSW
; intrpt.c    5          x++;
        INC     _x+1
        MOV     A,_x+1
        JNZ     _3
        INC     _x
_3:
; intrpt.c    6          y += x-3;
        MOV     R7,_x+1
        MOV     R6,_x
        MOV     R5,#03H
        MOV     R4,#00H
        LCALL   __MINI
        MOV     R0,#_y
        LCALL   __CAPLIID
; intrpt.c    7
; intrpt.c    8          return;
; intrpt.c    9 }
        POP     PSW
        POP     AR7
        POP     AR6
        POP     AR5
        POP     AR4
        POP     AR0
        POP     ACC
        RETI

; intrpt.c   10

        EXTRN   CODE(__MINI)
        EXTRN   CODE(__CAPLIID)
        EXTRN   CODE(SMALL)
        END
```

When an interrupt occurs, the vector instructs the processor to jump to the handler. The interrupt handler always saves PSW. When the `_using()` qualifier is being used it will switch to the correct register bank by loading a new value in PSW, based on the value specified with the `_using()` qualifier. When this qualifier is not being used each of the registers R0–R7 which are (or could be) used in the interrupt routine will be saved. Each of the non R0–R7 registers: A,B and DPTR, are also being saved when they are used in the routine. After the context is being saved the user C interrupt function is being executed, and when it is completed the context is being restored. All saved registers are being popped from the stack including PSW. By restoring the original PSW value, the correct register bank is being restored automatically. Finally the RETI (return from interrupt) is executed.

In the above example the C interrupt function uses the following registers: A, R0, R4, R5, R6, R7 and PSW. All of these registers are being saved on function entry and restored on function exit. When the using() qualifier would have been used the compiler would omit saving/restoring register R0 and R4–R7, but instead code would be generated to switch the register bank (e.g. `MOV PSW,#`18 for register bank 3).

Because the PUSH and POP instruction require a direct address operand, the assembler uses the predefined symbolic register addresses AR0–AR7 to push and pop the corresponding registers R0–R7.

The relation between the interrupt number and the vector address is: *interrupt_id* = (*vector_address* – 3)/8. In the example above, where the interrupt number is `17`, the vector address is `08BH`.

You can write your own interrupt handler (and interrupt vector) in assembly. When you don't want the vector to be generated automatically you can use the **–v** command (or **#pragma novector**). When you don't want the interrupt frame (saving/restoring registers) to be generated you can use the **–vf** command. In that case you will have to specify your own interrupt frame. For this you can use the inline capabilities of the compiler. The example below shows an interrupt function for which only DPTR has to be saved and restored.

Example:

```
; 8051 C compiler vx.y rz        SNaaaa (c) year TASKING, Inc.
; options: -s -vf
$CASE
        NAME    INT
; intrpt.c       1       _inline _using(1) void
; intrpt.c       2       interrupt_prolog( void )
```

```
; intrpt.c       3      {
; intrpt.c       4      #pragma asm
; intrpt.c       5              PUSH DPL
; intrpt.c       6              PUSH DPH
; intrpt.c       7      #pragma endasm
; intrpt.c       8      }
; intrpt.c       9
; intrpt.c      10      _inline _using(1) void
; intrpt.c      11      interrupt_epilog( void )
; intrpt.c      12      {
; intrpt.c      13      #pragma asm
; intrpt.c      14              POP DPH
; intrpt.c      15              POP DPL
; intrpt.c      16      #pragma endasm
; intrpt.c      17      }
; intrpt.c      18
; intrpt.c      19      _bit int1_flag;
        PUBLIC  _int1_flag
C51_BI  SEGMENT BIT
        RSEG    C51_BI
_int1_flag:     DBIT    1
; intrpt.c      20
; intrpt.c      21      _interrupt(1) _using(1) void
; intrpt.c      22      alarm( void )
; intrpt.c      23      {
        PUBLIC  _?alarm
        CSEG AT 0BH
        JMP     _?alarm
; free registers in this function: A B DPTR R0 R1 R2 R3 R4 R5 R6 R7
INT_ALARM_PR    SEGMENT CODE
        RSEG    INT_ALARM_PR
_?alarm:
        USING   1
; intrpt.c      24              interrupt_prolog();
        PUSH DPL
        PUSH DPH
; intrpt.c      25
; intrpt.c      26              int1_flag = 1;
        SETB    _int1_flag
; intrpt.c      27
; intrpt.c      28              interrupt_epilog();
        POP DPH
        POP DPL
; intrpt.c      29      }
        RETI

; intrpt.c      30

        EXTRN   CODE(SMALL)
        END
```

Pragma intsave

When using assembly in an interrupt function, it might be necessary to save registers not being saved automatically by the compiler. For this you can use **#pragma intsave** *registers*.

Example:

```
#pragma intsave A R0 R1
/* the interrupt function uses registers A, R0 and R1
 */

_interrupt(1) void alarm( void )
{
#pragma asm
        MOV A,0C8H
        XCH A,R1
        MOV A,0C9H
        ADD A,R1
        MOV 0C9H,A
#pragma endasm
```

If an interrupt function does not use the fast parameter area, you can instruct the compiler to generate a shorter interrupt frame with **#pragma intsave NOPARM***regbank*. With this pragma the compiler does not generate code to save and restore the present data in the fast parameter area.

Example:

```
#pragma intsave NOPARM0
```

With the option **–nofastparm** you can instruct the compiler to switch off the use of the fast parameter section for all functions. In this case you do not need **#pragma intsave NOPARM***regbank*.

_frame function qualifier

With the `_frame` function qualifier you can specify which registers and SFRs must be saved for a particular interrupt function. Only the specified registers will be pushed and popped from the stack. The syntax is:

_frame(*reg*[,*reg*]... **)**

Example:

```
_interrupt(1) _frame(A,R0,R1) void alarm( void )
{
     /* an interrupt function */
}
```

Pragma vector

For certain ROM monitors it is necessary to specify an offset for all interrupt vectors. For this you can use the command **–ivo=***value* or **#pragma VECTOR** *value*. Suppose the previous example is built for a ROM monitor with the interrupt table at offset 0x4000. When compiling the example with **–ivo=0x4000** the vector is being located at address 0x400B instead of 0xB.

7.8 MULTIPLE DATA POINTER SUPPORT

The standard 8051 architecture provides just one 16–bit pointer for indirect addressing of external memory (DPTR). At this moment there are several architectures supporting more than just one data pointer. The Infineon Technologies C500/C800 family has support for 8 16–bit data pointers, the Dallas 80C320/520/530 and AMD 80C521 have support for 2 16–bit data pointers, and also the Philips 51 family has support for 2 16–bit data pointers. Using more than one data pointer is mainly useful when copying bytes from source to destination or when comparing different areas in memory. Most beneficial for multiple data pointer optimization is therefore the C library containing a lot of functions in that area. The table below shows which 8051 C functions benefit from using multiple data pointers.

MODEL	Small	Aux	Large	Reentrant
strcmp()			x	x
strcpy()			x	x
strncmp()			x	x
strncpy()			x	x
memcmp()			x	x
memcpy()			x	x
memmove()			x	x
xdxdcpy()	x	x	x	x
xdxdmove()	x	x	x	x
romxdcpy()	x	x	x	x
romxdmove()	x	x	x	x

Table 7–3: Functions that benefit from multiple data pointers

All these C functions have been fully optimized for multiple data pointer support. These optimized functions can be used by linking the multiple data pointer library (**mdptr**[**dps**][**salr**]) before linking the standard C library. That way when using one of the functions the one using multiple data pointers will be linked instead of the standard implementation.

When using multiple data pointers in combination with interrupt functions it is necessary to make sure all data pointers are being saved and restored by the interrupt function. For the Infineon Technologies C500/C800 family use the command **–ps**, for the Dallas 80C320/520/530, AMD 80C521 use the command **–pd** and for the Philips 51 family use **–pp** in order to generate appropriate interrupt frames. The example below shows the interrupt frame for an interrupt function when using dual data pointer support (Dallas 80C320/520/530, AMD 80C521).

```
; 8051 C compiler vx.y rz       SNaaaa (c) year TASKING, Inc.
; options: -s -pd
$CASE
        NAME    INTRPT
; intrpt.c    1 _bit int1_flag;
        PUBLIC  _int1_flag
C51_BI  SEGMENT BIT
        RSEG    C51_BI
_int1_flag:     DBIT    1
; intrpt.c    2
; intrpt.c    3 _interrupt(1) _using(1) void
; intrpt.c    4 alarm( void )
; intrpt.c    5 {
        PUBLIC  _?alarm
        CSEG AT 0BH
        JMP     _?alarm
; free registers in this function: A B DPTR R0 R1 R2 R3 R4 R5 R6 R7
INTRPT_ALARM_PR SEGMENT CODE
        RSEG    INTRPT_ALARM_PR
_?alarm:
        USING   1
        PUSH    ACC
        PUSH    B
        PUSH    DPL
        PUSH    DPH
        PUSH    084H
        PUSH    085H
        PUSH    086H
        MOV     086H,#00H
        PUSH    PSW
        MOV     PSW,#08H
; intrpt.c    6         int1_flag = 1;
        SETB    _int1_flag
; intrpt.c    7 }
        POP     PSW
        POP     086H
        POP     085H
        POP     084H
        POP     DPH
        POP     DPL
        POP     B
        POP     ACC
        RETI
```

```
; intrpt.c    8

        EXTRN   CODE(SMALL)
        END
```

7.9 ASSEMBLY LANGUAGE INTERFACING

Assembly language functions can be called from C–51 and vice versa. The names used by **cc51** are case sensitive, so you must tell **asm51** to act case sensitive too, using the $CASE control. **cc51** prepends an underscore for the name of the C variable, to distinguish these names from the 8051 registers. So, any names used or defined in C–51 must have a leading underscore in assembly code. Internal compiler symbols (run–time library) use two underscores.

The assembler uses the following naming convention for C variables and functions:

Name in C	Name used in assembly
variable	_variable
_cdecl function()	_function
function()	_?function
_regparm function()	_?function
_regparm function(...) /* function with variable argument list */	_??function

Table 7–4: Naming convention for variables and functions

When you call an assembly routine that has a name of e.g. 50 characters, you get a link error "UNRESOLVED EXTERNAL". The reason for it is that the C compiler truncates names to 32 characters, but the assembler and linker do not. The solution is, when calling assembly routines, use names of 31 characters or less (if you do not count the leading '_' for a moment). The same rule applies when you call a C function from your assembly code.

The following parameter passing scheme is used:

1. For functions declared `_regparm` (default), the first non bit arguments are passed via registers; the __PARM*x* area will not be used by these functions. This register parameter passing scheme is memory model independent.

2. Parameters which do NOT fit in the register passing scheme, are passed the same way as done by `_cdecl`.

3. For `_small`, `_aux` and `_large` functions having the `_cdecl` qualifier, the data locations for function parameters are in data fields with the same name as the function itself (also prepended with an underscore), but with `_BIT` or `_BYTE` appended to it. An assembly function with parameters must define those data fields in a XDAT or DATA segment, depending on the memory model used with the C modules. Of course, bit parameters must always be defined in a BIT segment.

   For `_reentrant` functions, `_cdecl` parameter passing is done using the virtual stack.

4. For `_aux` and `_large` functions, when using `_cdecl` new–style prototypes, the compiler tries to pass the first arguments in a static field in data, called __PARM*x*, where *x* is the register bank used. To simplify the programming of an assembly routine, prototype the routine with a `_cdecl` qualifier. Now all parameters will simply be passed using the *module_function*_BYTE area. However, if all parameters fit in the register parameter passing scheme, `_regparm` is recommended.

For more information on parameter passing see section 3.4, *Function Parameters* in chapter *Language Implementation*.

The quickest (and most reliable) way to make an assembly language function, which must conform to C–51, is to make the body of this function in C, and compile this module with the memory model used by all other C modules. If the assembly function must return something, specify the return type in the 'assembler function' using C syntax, and let it return something. If parameters are used, force code generation for accessing these parameters with a dummy statement (e.g. an assignment) or declare the parameter as volatile and just access it:

```
int assem( char volatile a, char c, int i )
{
     a;
     return( c + i );
}
```

Now compile this module, using the correct memory model. The compiler makes the correct frame, and you can edit the generated assembly module, to make the real assembly function inside this frame.

For more information on return types see section 7.2, *Register Usage* in this chapter.

A second method to create an interface to assembly is to make use of the feature of the **cc51** compiler to have inline assembly.

Assembly lines in the C–source must be introduced by a '#pragma asm', the end is indicated by a '#pragma endasm'. For example:

```
int assem( char c, int i )
{
     int j;
     j = i;
#pragma asm
     MOV P2,#01
#pragma endasm
     j = c;
}
```

When the assembly does not change any registers, like in the example above, also '#pragma asm_noflush' may be used instead of '#pragma asm'.

For an explanation of the used pragmas see section 4.4, *Pragmas*.

7.10 REENTRANT MODEL / _REENTRANT FUNCTIONS

When you use the reentrant model (**–Mr** option) or some `_reentrant` functions, a virtual stack mechanism is used. A special stack pointer to this virtual stack is made. Non register function parameters are pushed on the virtual stack and removed after the function call.

During these actions the virtual stack pointer is updated more than once. This operation however needs several instructions. When a program uses interrupts, it is very well possible that an interrupt occurs during the update of the virtual stack pointer. The not yet correct virtual stack pointer will be changed and is thus pointing to an undefined address.

You can use the standard library as delivered with the compiler when the virtual stack is NOT accessed during the interrupt. This is guaranteed if:

– the C interrupt function is `_small`, `_aux` or `_large`

  and

– the C interrupt function is not calling (direct or indirect) a `_reentrant` function

In all other cases, the update of the virtual stack pointer should prevent interrupts to occur. This can be done by disabling and enabling the interrupts during the update. However, this will slow down the program and increase the interrupt response time. Therefore, the default libraries delivered with the compiler (`c51s.lib`, `c51a.lib`, `c51m.lib` or `c51r.lib`) do not disable interrupts during a virtual stack pointer update. Special protected libraries are delivered for this purpose (`c51sp.lib`, `c51ap.lib`, `c51mp.lib` or `c51rp.lib`). So, when linking replace the normal C library with the protected version.

In EDE you can select a protected library by enabling the options `Application uses reentrant functions` and `Use protection on virtual stack pointer updates` in the `Linker | Stack/Heap` entry of the `Project | Project Options...` dialog.

7.11 LINKING AN APPLICATION

This section explains how to link your C–51 application.

A typical linker command for a C–51 application looks like this:

```
link51 cstart.obj,your_objects, libraries to
output_name options
```

Note that the file `cstart.obj` must also be linked. For the small model, a default `cstart.obj` is delivered. When you use another model, or when you want something specific (e.g. when using 'malloc()', specify a heap), you have to create your own `cstart.obj`.

You have to make your own `cstart.obj` when:

- You use another model than the small model.
- You use 'malloc()', 'calloc()' in your application.
- You use floating point in your application.
- You want memory to be cleared on startup (i.e. 'static' objects should have value '0' at startup).
- You want to run the application in another register bank than bank '0'.
- You want to have a larger stack size in the reentrant model.
- You need to use a protected version of the reentrant model.

- You need to have a virtual stack.

Apart from the startup code, you have to specify all your own object files and libraries. The last objects to link are the C library delivered with the C–51 package. For each model a specific library is delivered, you have to choose the one you need (see chapter 6, *Libraries*). If you use the reentrant model, see also section 7.10, *Reentrant Model / _reentrant Functions*.

When you have linked the wrong C library, you get an unresolved external during the link phase. The external has the name of the model you used in your application. I.e. 'SMALL', 'AUX', you will get such an unresolved external.

When you have used floating point within your application, you have to link the floating point run–time library too. This library must be placed after the C–51 library. So,

```
link51 cstart.obj,my.obj,float.lib,c51s.lib to my.out
```

does NOT work (probably unresolved externals will be the result). But:

```
link51 cstart.obj,my.obj,c51s.lib,float.lib to my.out
```

is correct.

As an *option* to the linker, the option 'FUNCTIONOVERLAY' (or 'FO') must be specified, unless the majority of the application consists of PL/M instead of C. With this option you specify to the linker that it should overlay as much local C data as possible, thus saving data space.

If you specify to the linker that it may overlay data, you have to specify all indirect function calls you have used in your application (i.e. all functions which are called by using function pointers). How to do this can be found in the user manual of **link51**.

7.12 TROUBLESHOOTING

This section describes a number of commonly made mistakes and what you can do about them.

7.12.1 LINKING PROBLEMS

Problem: Unresolved externals are found, among the symbols is one (or more) of the names 'SMALL', 'AUX', 'LARGE' or 'REENTRANT'.

Possible causes:

No C library is specified on the linker command line.

The wrong C library is linked.

The file `cstart.obj` is made for the wrong model.

One of the objects specified is compiled in the wrong model.

Problem: Unresolved externals are found, names end on `_BYTE` or `_BIT`.

Possible cause:

Prototypes of functions are not present or do not match with the function definition. These types of errors are detected by the compiler. See also `_cdecl` and `_regparm` function qualifiers (**–OR** and **–Or**).

Problem: Unresolved externals are found, names '__HEAPSTART' and '__HEAPEND' are in between.

Cause: No heap space is specified in the C startup code, the heap is needed for 'malloc()', 'calloc()', 'realloc()'.

Problem: Unresolved externals are found, names are found which do not occur in the application.

Cause: Does the application use floating point arithmetic? Maybe the library `float.lib` is not linked.

Floating point arithmetic is used, but the floating point library `float.lib` is not specified AFTER the C library. See also section 7.11, *Linking an Application*.

Problem: Unresolved externals are found, names '__FLOATSTART' and '__FLOATEND' are in between.

Cause: The application uses floating point, but no floating point stack is specified within the `cstart` module.

Problem: Unresolved externals are found. A name like '*_function*' is in between.

Cause: The C–application uses the function named *function*, but the function itself is not linked or is not programmed as a `_cdecl` function (see also the **–Or** option). Check the function prototypes used in the application.

Problem: Unresolved externals are found. A name like '*_?function*' is in between.

Cause: The C–application uses the function named *function*, but the function itself is not linked or is not programmed as a `_regparm` function (see also the **–Or** option). Check the function prototypes used in the application.

Problem: Unresolved externals are found. A name like '*_??function*' is in between.

Cause: The C–application uses the function named *function*, but the function itself is not linked or is not programmed as a `_regparm` function (see also the **–Or** option), or the function is not programmed as a variable argument function. Check the function prototypes used in the application.

7.12.2 RUN-TIME PROBLEMS

Problem: Variables and parameters of one procedure are overwritten by another procedure.

Cause: Prototypes of functions do not match their function definition. These types of errors are detected by the compiler.

The application uses function pointers, when calling a function indirectly (using a function pointer), the linker is not aware of this call. You have to specify these calls with the 'FUNCTIONOVERLAY' control, otherwise data is illegally overlaid.

Problem: Variables and parameters of some function are overwritten.

Cause: In static models, functions may never be recursive. The C–51 compiler cannot check this. Use the 'FUNCTIONOVERLAY' option of the linker together with the 'GRAPH()' option. **Link51** checks if recursion is found in the application, reports the recursion in a function call graph and refuses to continue.

Problem: Variables are not '0' during startup of the program.

Cause: In the C language it is defined that 'static' variables, which are not initialized at startup, have the value '0'. In C–51 you have to specify this in the startup code. Another way to get around this problem is to initialize the variables with the value '0'.

Problem: Interrupts are not disabled at startup of the program.

Cause: By default, the C startup code will not clear the enable all (EA) bit. When you define the preprocessor symbol CLR_EA all interrupts are disabled at startup. In EDE you can define this symbol by enabling the option `Disable all interrupts at startup` in the `Processor | Startup Code` entry of the `Project | Project Options...` dialog.

APPENDIX A

MISRA C

TASKING

APPENDIX

Supported and unsupported MISRA C rules

x means that the rule is not supported by the TASKING C compiler.
(R) is a required rule, (A) is an advisory rule.

	1.	(R)	The code shall conform to standard C, without language extensions
x	2.	(A)	Other languages should only be used with an interface standard
	3.	(A)	Inline assembly is only allowed in dedicated C functions
x	4.	(A)	Provision should be made for appropriate run–time checking
	5.	(R)	Only use characters and escape sequences defined by ISO C
x	6.	(R)	Character values shall be restricted to a subset of ISO 106460–1
	7.	(R)	Trigraphs shall not be used
	8.	(R)	Multibyte characters and wide string literals shall not be used
	9.	(R)	Comments shall not be nested
x	10.	(A)	Sections of code should not be ”commented out”
	11.	(R)	Identifiers shall not rely on significance of more than 31 characters
	12.	(A)	The same identifier shall not be used in multiple name spaces
	13.	(A)	Specific–length typedefs should be used instead of the basic types
	14.	(R)	Use ’unsigned char’ or ’signed char’ instead of plain ’char’
x	15.	(A)	Floating point implementations should comply with a standard
x	16.	(R)	The bit representation of floating point numbers shall not be used
	17.	(R)	”typedef” names shall not be reused
x	18.	(A)	Numeric constants should be suffixed to indicate type
	19.	(R)	Octal constants (other than zero) shall not be used
	20.	(R)	All object and function identifiers shall be declared before use

	21.	(R)	Identifiers shall not hide identifiers in an outer scope
	22.	(A)	Declarations should be at function scope where possible
x	23.	(A)	All declarations at file scope should be static where possible
	24.	(R)	Identifiers shall not have both internal and external linkage
x	25.	(R)	Identifiers with external linkage shall have exactly one definition
	26.	(R)	Multiple declarations for objects or functions shall be compatible
x	27.	(A)	External objects should not be declared in more than one file
	28.	(A)	The ”register” storage class specifier should not be used
	29.	(R)	The use of a tag shall agree with its declaration
	30.	(R)	All automatics shall be initialized before being used
	31.	(R)	Braces shall be used in the initialization of arrays and structures
	32.	(R)	Only the first, or all enumeration constants may be initialized
	33.	(R)	The right hand operand of && or || shall not contain side effects
	34.	(R)	The operands of a logical && or || shall be primary expressions
	35.	(R)	Assignment operators shall not be used in Boolean expressions
x	36.	(A)	Logical operators should not be confused with bitwise operators
	37.	(R)	Bitwise operations shall not be performed on signed integers
	38.	(R)	A shift count shall be between 0 and the operand width minus 1
	39.	(R)	The unary minus shall not be applied to an unsigned expression
	40.	(A)	”sizeof” should not be used on expressions with side effects
x	41.	(A)	The implementation of integer division should be documented
	42.	(R)	The comma operator shall only be used in a ”for” condition

	43.	(R)	Don't use implicit conversions which may result in information loss
	44.	(A)	Redundant explicit casts should not be used
	45.	(R)	Type casting from any type to or from pointers shall not be used
	46.	(R)	The value of an expression shall be evaluation order independent
	47.	(A)	No dependence should be placed on operator precedence rules
	48.	(A)	Mixed arithmetic should use explicit casting
	49.	(A)	Tests of a (non–Boolean) value against 0 should be made explicit
	50.	(R)	F.P. variables shall not be tested for exact equality or inequality
x	51.	(A)	Constant unsigned integer expressions should not wrap–around
	52.	(R)	There shall be no unreachable code
	53.	(R)	All non–null statements shall have a side–effect
	54.	(R)	A null statement shall only occur on a line by itself
	55.	(A)	Labels should not be used
	56.	(R)	The "goto" statement shall not be used
	57.	(R)	The "continue" statement shall not be used
	58.	(R)	The "break" statement shall not be used (except in a "switch")
	59.	(R)	An "if" or loop body shall always be enclosed in braces
	60.	(A)	All "if", "else if" constructs should contain a final "else"
	61.	(R)	Every non–empty "case" clause shall be terminated with a "break"
	62.	(R)	All "switch" statements should contain a final "default" case
	63.	(A)	A "switch" expression should not represent a Boolean case
	64.	(R)	Every "switch" shall have at least one "case"
	65.	(R)	Floating point variables shall not be used as loop counters
x	66.	(A)	A "for" should only contain expressions concerning loop control

x	67.	(A)	Iterator variables should not be modified in a ”for” loop
	68.	(R)	Functions shall always be declared at file scope
	69.	(R)	Functions with variable number of arguments shall not be used
	70.	(R)	Functions shall not call themselves, either directly or indirectly
	71.	(R)	Function prototypes shall be visible at the definition and call
	72.	(R)	The function prototype of the declaration shall match the definition
	73.	(R)	Identifiers shall be given for all prototype parameters or for none
	74.	(R)	Parameter identifiers shall be identical for declaration/definition
	75.	(R)	Every function shall have an explicit return type
	76.	(R)	Functions with no parameters shall have a ”void” parameter list
x	77.	(R)	An actual parameter type shall be compatible with the prototype
	78.	(R)	The number of actual parameters shall match the prototype
	79.	(R)	The values returned by ”void” functions shall not be used
	80.	(R)	Void expressions shall not be passed as function parameters
x	81.	(A)	”const” should be used for reference parameters not modified
	82.	(A)	A function should have a single point of exit
	83.	(R)	Every exit point shall have a ”return” of the declared return type
	84.	(R)	For ”void” functions, ”return” shall not have an expression
	85.	(A)	Function calls with no parameters should have empty parentheses
x	86.	(A)	If a function returns error information, it should be tested
	87.	(R)	#include shall only be preceded by other directives or comments
	88.	(R)	Non–standard characters shall not occur in #include directives

	89.	(R)	#include shall be followed by either <filename> or ”filename”
	90.	(R)	Plain macros shall only be used for constants/qualifiers/specifiers
	91.	(R)	Macros shall not be #define’d and #undef’d within a block
	92.	(A)	#undef should not be used
x	93.	(A)	A function should be used in preference to a function–like macro
	94.	(R)	A function–like macro shall not be used without all arguments
x	95.	(R)	Macro arguments shall not contain pre–preprocessing directives
	96.	(R)	Macro definitions/parameters should be enclosed in parentheses
	97.	(A)	Don’t use undefined identifiers in pre–processing directives
	98.	(R)	A macro definition shall contain at most one # or ## operator
x	99.	(R)	All uses of the #pragma directive shall be documented
	100.	(R)	”defined” shall only be used in one of the two standard forms
	101.	(A)	Pointer arithmetic should not be used
	102.	(A)	No more than 2 levels of pointer indirection should be used
x	103.	(R)	No relational operators between pointers to different objects
	104.	(R)	Non–constant pointers to functions shall not be used
	105.	(R)	Functions assigned to the same pointer shall be of identical type
	106.	(R)	Automatic address may not be assigned to a longer lived object
x	107.	(R)	The null pointer shall not be de–referenced
x	108.	(R)	All struct/union members shall be fully specified
x	109.	(R)	Overlapping variable storage shall not be used
x	110.	(R)	Unions shall not be used to access the sub–parts of larger types
	111.	(R)	Bit fields shall have type ”unsigned int” or ”signed int”

	112.	(R)	Bit fields of type ”signed int” shall be at least 2 bits long
	113.	(R)	All struct/union members shall be named
	114.	(R)	Reserved and standard library names shall not be redefined
	115.	(R)	Standard library function names shall not be reused
x	116.	(R)	Production libraries shall comply with the MISRA C restrictions
x	117.	(R)	The validity of library function parameters shall be checked
	118.	(R)	Dynamic heap memory allocation shall not be used
	119.	(R)	The error indicator ”errno” shall not be used
	120.	(R)	The macro ”offsetof” shall not be used
	121.	(R)	<locale.h> and the ”setlocale” function shall not be used
	122.	(R)	The ”setjmp” and ”longjmp” functions shall not be used
	123.	(R)	The signal handling facilities of <signal.h> shall not be used
	124.	(R)	The <stdio.h> library shall not be used in production code
	125.	(R)	The functions atof/atoi/atol shall not be used
	126.	(R)	The functions abort/exit/getenv/system shall not be used
	127.	(R)	The time handling functions of library <time.h> shall not be used

See also section 3.18, *C Code Checking: MISRA C*, in Chapter *Language Implementation*.

APPENDIX B

SFR DEFINITION FILE

TASKING

3 APPENDIX

Next is an example of a Special Function Register (SFR) definition file, created for the 8051, 8031, 8751, 80C51, 80C31 and 87C51 derivatives. See the **–C***cpu* compiler option for a list of all SFR files.

/* special function register definition file: reg51.sfr */

_sfrbyte	P0	_at(0x80);	_sfrbit	TR1	_at(0x8E);
_sfrbyte	SP	_at(0x81);	_sfrbit	TF1	_at(0x8F);
_sfrbyte	DPL	_at(0x82);	_sfrbit	P1_0	_at(0x90);
_sfrbyte	DPH	_at(0x83);	_sfrbit	P1_1	_at(0x91);
_sfrbyte	PCON	_at(0x87);	_sfrbit	P1_2	_at(0x92);
_sfrbyte	TCON	_at(0x88);	_sfrbit	P1_3	_at(0x93);
_sfrbyte	TMOD	_at(0x89);	_sfrbit	P1_4	_at(0x94);
_sfrbyte	TL0	_at(0x8A);	_sfrbit	P1_5	_at(0x95);
_sfrbyte	TL1	_at(0x8B);	_sfrbit	P1_6	_at(0x96);
_sfrbyte	TH0	_at(0x8C);	_sfrbit	P1_7	_at(0x97);
_sfrbyte	TH1	_at(0x8D);	_sfrbit	RI	_at(0x98);
_sfrbyte	P1	_at(0x90);	_sfrbit	TI	_at(0x99);
_sfrbyte	SCON	_at(0x98);	_sfrbit	RB8	_at(0x9A);
_sfrbyte	SBUF	_at(0x99);	_sfrbit	TB8	_at(0x9B);
_sfrbyte	P2	_at(0xA0);	_sfrbit	REN	_at(0x9C);
_sfrbyte	IE	_at(0xA8);	_sfrbit	SM2	_at(0x9D);
_sfrbyte	P3	_at(0xB0);	_sfrbit	SM1	_at(0x9E);
_sfrbyte	IP	_at(0xB8);	_sfrbit	SM0	_at(0x9F);
_sfrbyte	PSW	_at(0xD0);	_sfrbit	P2_0	_at(0xA0);
_sfrbyte	ACC	_at(0xE0);	_sfrbit	P2_1	_at(0xA1);
_sfrbyte	B	_at(0xF0);	_sfrbit	P2_2	_at(0xA2);
_sfrbit	P0_0	_at(0x80);	_sfrbit	P2_3	_at(0xA3);
_sfrbit	P0_1	_at(0x81);	_sfrbit	P2_4	_at(0xA4);
_sfrbit	P0_2	_at(0x82);	_sfrbit	P2_5	_at(0xA5);
_sfrbit	P0_3	_at(0x83);	_sfrbit	P2_6	_at(0xA6);
_sfrbit	P0_4	_at(0x84);	_sfrbit	P2_7	_at(0xA7);
_sfrbit	P0_5	_at(0x85);	_sfrbit	EX0	_at(0xA8);
_sfrbit	P0_6	_at(0x86);	_sfrbit	ET0	_at(0xA9);
_sfrbit	P0_7	_at(0x87);	_sfrbit	EX1	_at(0xAA);
_sfrbit	IT0	_at(0x88);	_sfrbit	ET1	_at(0xAB);
_sfrbit	IE0	_at(0x89);	_sfrbit	ES	_at(0xAC);
_sfrbit	IT1	_at(0x8A);	_sfrbit	EA	_at(0xAF);
_sfrbit	IE1	_at(0x8B);	_sfrbit	P3_0	_at(0xB0);
_sfrbit	TR0	_at(0x8C);	_sfrbit	RXD	_at(0xB0);
_sfrbit	TF0	_at(0x8D);	_sfrbit	P3_1	_at(0xB1);

```
_sfrbit  TXD   _at( 0xB1 );
_sfrbit  INT0  _at( 0xB2 );
_sfrbit  P3_2  _at( 0xB2 );
_sfrbit  INT1  _at( 0xB3 );
_sfrbit  P3_3  _at( 0xB3 );
_sfrbit  P3_4  _at( 0xB4 );
_sfrbit  T0    _at( 0xB4 );
_sfrbit  P3_5  _at( 0xB5 );
_sfrbit  T1    _at( 0xB5 );
_sfrbit  P3_6  _at( 0xB6 );
_sfrbit  WR    _at( 0xB6 );
_sfrbit  P3_7  _at( 0xB7 );
_sfrbit  RD    _at( 0xB7 );
_sfrbit  PX0   _at( 0xB8 );
_sfrbit  PT0   _at( 0xB9 );
_sfrbit  PX1   _at( 0xBA );
_sfrbit  PT1   _at( 0xBB );
_sfrbit  PS    _at( 0xBC );
_sfrbit  P     _at( 0xD0 );
_sfrbit  OV    _at( 0xD2 );
_sfrbit  RS0   _at( 0xD3 );
_sfrbit  RS1   _at( 0xD4 );
_sfrbit  F0    _at( 0xD5 );
_sfrbit  AC    _at( 0xD6 );
_sfrbit  CY    _at( 0xD7 );
_sfrbit  ACC_0 _at( 0xE0 );
_sfrbit  ACC_1 _at( 0xE1 );
_sfrbit  ACC_2 _at( 0xE2 );
_sfrbit  ACC_3 _at( 0xE3 );
_sfrbit  ACC_4 _at( 0xE4 );
_sfrbit  ACC_5 _at( 0xE5 );
_sfrbit  ACC_6 _at( 0xE6 );
_sfrbit  ACC_7 _at( 0xE7 );
_sfrbit  B_0   _at( 0xF0 );
_sfrbit  B_1   _at( 0xF1 );
_sfrbit  B_2   _at( 0xF2 );
_sfrbit  B_3   _at( 0xF3 );
_sfrbit  B_4   _at( 0xF4 );
_sfrbit  B_5   _at( 0xF5 );
_sfrbit  B_6   _at( 0xF6 );
_sfrbit  B_7   _at( 0xF7 );
```

APPENDIX C

RESTRICTIONS FOR THE 80751 AND THE 80752

TASKING

9 APPENDIX

The 8xC751 and 8xC752 processors do not allow usage of the LCALL/LJMP and MOVX instructions. The following actions should be taken when you are not using EDE:

- Always call the assembler **asm51** with the SMALLROM and NOEXTERNALMEMORY controls.

  The SMALLROM control translates all LCALL/LJMP to ACALL/AJMP instructions and the NOEXTERNALMEMORY control issues an error when a MOVX instruction is encountered.

- Always use the **–rs** compiler option to tell the compiler that it must generate ACALL/AJMP instructions instead of LCALL/LJMP instructions.

With EDE, the assembler and compiler are automatically called with the controls and option mentioned above when you select the 8xC751 or 8xC752 processor.

The C library is delivered as `c751s.lib`. It contains **no** floating point functions, because floating point needs `xdat` memory. It also contains **no** run–time routines using `pdat` or `xdat` (not possible on this type of processor).

The floating point library is not supported due to the very limited resources of the 751 (only 64 bytes of RAM).

APPENDIX D

CONVERTING PL/M–51 APPLICATIONS TO C–51

TASKING

D APPENDIX

1 INTRODUCTION

This appendix describes some reasons why you should rewrite your PL/M–51 program within C–51. Then it describes how to convert an existing application written in PL/M–51 to C–51. It is noted what the difficulties are when converting to C–51. This appendix is not meant to learn the C language to a PL/M programmer.

2 WHY CONVERTING TO C-51

There has to be at least one good reason to rewrite a program into another language. This section contains a number of reasons why an existing application written in the PL/M–51 language should be rewritten in C–51.

- C applications are portable, switching to another processor in the future can be done just by recompiling your sources. Only target dependent portions of the program have to be rewritten.
- All programming constructs from the PL/M–51 language can be rewritten to C–51, except for the 'based variable' principle. Using pointers within C will normally overcome this problem.
- C–51 has more different memory models. Each memory model uses a different type of memory (data, pdat, xdat) to pass the procedure parameters. With PL/M–51, you have no choice, variables are always passed using internal RAM. This restricts the application in using variables.
- C–51 delivers a large set of standard library procedures, while PL/M–51 only has a few.
- C–51 produces code which is as small as, and most of the time even smaller in size than, code generated by the PL/M–51 compiler. Code also executes faster.
- C–51 supports floating point. PL/M–51 has no floating point arithmetic available within the language.
- C–51 has a reentrant model in which you can write recursive programs. PL/M–51 does not have such a model.
- Overlaying of data (parameters/automatics) within C–51 is done on function/procedure base. Therefore it is not needed to place all code within one source module to get the best overlaying results.

- C–51 is able to work together with PL/M–51 modules. The main procedure of the program MUST be written within C. Overlaying data of PL/M–51 modules with C–51 procedures is not possible. Mixing C–51 and PL/M–51 may cause a less optimal usage of data memory.

3 POINTS OF ATTENTION

From here we will note a number of things you should take care of, when converting your PL/M–51 source to C–51. Constructions not noted here are really straightforward to convert between the two languages. You only need to have little knowledge of the PL/M–51 language and the C language to be able to convert those constructions.

Names and identifiers

Within C, the '$' character is a real character within an identifier. In PL/M–51 this character is ignored when comparing names of identifiers.

Data types

Basic data types of PL/M–51 do also exist within C–51. Note that all data types within PL/M–51 are unsigned types. Within C–51, you have to specify a variable to be unsigned, while default types are signed.

Constants

C–51 does not have a notation for binary numbers. Instead you have to use decimal, octal or hexadecimal notation.

Variables

A variable cannot be 'AT'ed at another variable within C–51. Instead you should use union variable types. Declaring a struct of eight bits on the same address as a character can be done with use of the `_bitbyte` type variable.

Placing a variable on an absolute address is done with the `_at()` attribute.

Special function registers are NOT declared using the `register` keyword. You can place your own special function registers using the `_sfrbit` and `_sfrbyte` data types.

'BASED' variables is in fact the same as using pointers from within C–51. Pointers to bit variables do not exist.

Procedures

Procedures cannot be declared as nested procedures. Hiding the occurrence of a procedure to another source module can be done using the 'static' attribute.

Type conversions

Type conversions within C–51 are different than in PL/M–51. In C–51, whenever an expression contains an integer and a character, the character is always converted to an integer. I.e. an unsigned character has a high byte value of 0, a signed character will get sign extension in its high byte.

Expressions containing bit variables, together with other types of variables are allowed within C–51. The bit variable will be converted to the type required, the result of the conversion is the value 0 or 1.

Statements

The 'DO WHILE' of PL/M–51 is the same as the 'while' within C–51. Do NOT use the 'do while' construction within C–51 for this, because the test is done afterwards, the loop will always be executed at least once.

You cannot program a GOTO from one procedure to another.

Indirect procedure call

Like in PL/M–51, within C–51 you cannot transfer parameters to a procedure called indirectly. The one exception is when programming in the reentrant model of C–51. Also like PL/M–51, the linker will not notice indirectly called procedures, therefore you have to specify to the linker which procedures do call each other indirectly, otherwise the linker will overlay the parameter and local space of these procedures. This results in incorrect execution behavior of the program.

It is not possible to specify a procedure to be called indirectly.

PL/M–51 built–in procedures

C–51 does not have the built–in procedures like PL/M–51. However, most of the procedures can easily be simulated using some macro definitions. Here the macro definitions follow, which you can use within the C–program.

```
#define LENGTH(x)      (sizeof(x)/sizeof(x[0]))
#define LAST(x)        (LENGTH(x)-1)
#define SIZE(x)        (sizeof(x))
#define LOW(x)         ((unsigned char)(x))
#define HIGH(x)        ((unsigned char)((x) >> 8))
#define DOUBLE(x)      ((unsigned int)(x))
#define BOOLEAN(x)     ((x) & 0x01)
#define EXPAND(x)      ((unsigned char)(x))
#define PROPAGATE(x)   ((unsigned char)(0-(x)))
#define SHL(x,y)       ((x) << (y))
#define SHR(x,y)       ((x) >> (y))
#define DEC(x)         (_da((x) -= 1))
```

For 'ROL' and 'ROR' you can use the built–in C–51 procedures '_rol' and '_ror'. However these can only be used on character type variables. There is no such procedure for integer type variables.

Instead of 'TESTCLEAR' you should use the built–in procedure '_testclear' within C–51.

C–51 has no alternative to the 'TIME' procedure. Also no alternatives are present to the 'SCL' and 'SCR' procedures. Because these procedures are not easily programmed within C (or they will need a lot of code), these procedures can best be written within assembly (or inline assembly within the C–program).

Note the 'DEC' macro as written above uses the inline procedure '_da()' to do the decimal adjust.

4 USING PL/M-51 TOGETHER WITH C-51

It is possible to mix C–51 code with PL/M–51 code. Thus you are able to rewrite parts of the program within C–51, while other parts remain written within PL/M–51.

One restriction has to be noticed. When linking your PL/M–51 application, normally the option OVERLAY is used. This option will overlay data of modules which do not call each other. You have to specify to the linker which modules do call each other indirectly.

When creating an application within C–51, normally the option FUNCTIONOVERLAY is used to the linker. This option will overlay data of procedures which do not call each other. Now you have to specify which procedures call each other indirectly.

Because these two overlaying mechanisms are incompatible. When mixing PL/M–51 with C–51 code you have to choose one of the overlaying mechanisms. This results in less optimal data overlaying, because data of C–51 modules will never be overlayed with data of PL/M–51 modules. Thus probably more data space will be necessary in relation to the application being completely written in one language.

APPENDIX E

CPU FUNCTIONAL PROBLEMS

TASKING

APPENDIX

1 INTRODUCTION

Several chip suppliers publish microcontroller errata sheets for reporting both functional problems and deviations from the electrical and timing specifications.

For some of these functional problems in the microcontroller itself, TASKING's 8051 C compiler and/or assembler can provide workarounds. In fact these are software workarounds for hardware problems.

This appendix lists a summary of functional problems which can be bypassed by the compiler tool kit.

Please refer to the chip supplier's errata sheets to verify if you need to use one of these bypasses.

2 CPU FUNCTIONAL PROBLEM BYPASSES

Generate NOP instruction before DIV AB instruction

Dallas reference: Dallas DS80C390 erratum #6, revision B3 01/19/00

Use compiler option:

–bp1

or use assembler control:

BYPASS(1)

When you use the Dallas DS80C390 derivative, the DIV AB instruction may return erroneous results if the A register is accessed immediately preceding the DIV AB instruction.

With the compiler option **–bp1** (or the assembler control BYPASS(1)) an extra NOP is inserted before any DIV AB instruction.

APPENDIX F

MIGRATION FROM KEIL, FRANKLIN OR ARCHIMEDES

TASKING

APPENDIX

1 INTRODUCTION

This appendix explains how you can migrate your C–51 application from the Keil, Franklin or Archimedes C–51 compiler to the TASKING C–51 compiler (**cc51**).

There are two major areas of differences between the two compilers which are the cause of most of the changes you will have to deal with. First of all the C language extensions present in both compilers to support special 8051 family features. And secondly the compiler output. The Keil compiler produces an object file as output whereas the TASKING compiler generates an assembler file. The Keil compiler uses controls to steer the behavior of the compiler. The TASKING compiler behavior is steered with the more commonly used options, making command lines both shorter and clearer.

2 ANSI-C EXTENSIONS

In the TASKING compiler all extra keywords for the 8051 extensions of ANSI–C have an underscore prefix. The most frequently occurring differences in keywords between the Keil compiler and the TASKING compiler can be resolved by using preprocessor definitions. The header file `keil.h` contains these definitions. This file can be 'included' for all C source files using the command line option "**–Hkeil.h**".

2.1 MEMORY TYPE QUALIFIERS

Both compilers have the same memory type qualifiers, which are used in the same place in the grammar so a simple preprocessor definition suffices. For this purpose the definitions listed below are included in `keil.h`.

```
/* memory type keywords */

#define data   _data
#define bdata  _bdat
#define idata  _idat
#define pdata  _pdat
#define xdata  _xdat
#define code   _rom
```

2.2 POINTERS

The TASKING compiler does not support generic (3–byte) pointers, since it is considered to be too costly both in execution time and memory usage. Therefore, pointers declared without a specification of the referred memory type will point to the default memory type (which depends on the selected memory model) being either 1 or 2 bytes. This rule is also valid for library functions using pointers. Sometimes you might want to make an exception, e.g. to keep the format strings of the `printf()` function and related library functions in rom. This is also supported, as is explained in section 3.10 *Strings*.

2.3 ABSOLUTE VARIABLE ALLOCATION

The way of specifying the address for an absolute variable is also different for the two compilers, as is shown below.

```
/* Keil compiler */        /* TASKING compiler */

char var _at_ 0x80;        char var _at( 0x80 );
```

The TASKING way makes it easy to make a preprocessor define to convert the ANSI–C extension. This is particularly useful when you want to compile your program with a compiler for your host. Especially for this purpose the include file `cc51.h` has been included in the package.

2.4 SFR REGISTERS

As with absolute variable allocation the TASKING compiler uses a different way of specifying the address of an SFR register than the Keil compiler, as is shown in the following example:

```
/* Keil compiler */       /* TASKING compiler */

sfr PSW = 0xd0;           _sfrbyte PSW _at( 0xd0 );
sbit OV = PSW ^ 2;        _sfrbit  OV  _atbit( PSW, 2);
```

However, you will rarely need to specify an SFR register, because for most 8051 derivatives a register include file is delivered with the package.

The Keil **sfr16** type is supported by the TASKING compiler, however the Keil **sfr16** type implicitly handles SFR registers little endian, even though by default integers are handled big endian. To handle SFR words in little endian the TASKING compiler supports the specifier **_little**, so the Keil **sfr16** type equals the TASKING **_sfrword _little** type.

```
/* Keil compiler */      /* TASKING compiler */

sfr16 RCAP2 = 0xca;      _sfrword _little RCAP2 _at( 0xca );
```

2.5 FUNCTION QUALIFIERS

For the TASKING compiler the extra qualifiers for functions defined for the ANSI extension appear in the syntax *before* the function name and parameter list. This is conform the ANSI syntax for assigning type and other qualifiers to an object (variable or function). In contrast these attributes appear *after* the function name and parameter list in the syntax for the Keil compiler.

```
/* Keil compiler */

int func( char c ) small interrupt 1 using 2
{
     <function_code>
}

/* TASKING compiler */

_small _interrupt(1) _using(2) int func( char c )
{
     <function_code>
}
```

Note that for the TASKING compiler 'reentrant' is a separate model and not, as for the Keil compiler, a function attribute. So, for the migration from the Keil compiler to the TASKING compiler you should use the **_reentrant** model qualifier for every function with the **reentrant** qualifier. You should ignore any model qualifier you were using in combination with the **reentrant** qualifier:

```
/* Keil compiler */

int func( char c ) large reentrant

/* TASKING compiler */

_reentrant int func( char c )
```

The function qualifier used by both compilers to make parameter passing for a function according to the PL/M–51 convention is placed in both grammars before the function identifier. So, the preprocessor definition included in `keil.h`, as shown below, is sufficient for an easy migration on this point.

```
/* function qualifier keyword */

#define alien  _plmprocedure
```

2.6 ASSEMBLY INTERFACE

The passing of parameters to a function via registers is exactly the same in both compilers. The name and offset for parameters passed in memory is incomparable between the two compilers.

The two compilers use almost the same registers for the return value of a function, as shown in the following table.

Return Type	Keil Compiler	TASKING Compiler
bit	Carry	Carry
char / 1–byte pointer	R7	A
int / 2–byte pointer	R6–R7	R6–R7
long	R4–R7	R4–R7
float	R4–R7	Floating point stack
generic pointer	R1–R3	–

Table F–1: Function return types

2.7 BUILT-IN (INTRINSIC) FUNCTIONS

Both compilers have several built–in (intrinsic) functions. However, only four of them have the same objective. For these four a preprocessor definition, which is included in the header file `keil.h` as listed below, will establish the migration.

```
/* names inline functions */

#define _crol_      _rol
#define _cror_      _ror
#define _testbit_   _testclear
#define _nop_       _nop()
```

For the other built–in rotate functions of the Keil compiler, namely `_irol_`, `_lrol_`, `_iror_` and `_lror_`, there are no direct replacements.

The TASKING compiler has two other built–in functions namely `_getbit` and `_putbit`, which can be used to manipulate bits in an arbitrary bitaddressable byte without the need to declare a separate bit identifier for each bit.

2.8 LIBRARY ROUTINES

The prototypes of the library routines of the TASKING compiler package are conform to the ANSI standard. This is not the case for the libraries included in the Keil compiler package. However, in most of these cases the Keil library uses character parameters where the TASKING library uses integers, but due to automatic type conversions generated by the compiler this will usually not create problems.

Both compiler packages allow you to change the I/O mechanism for all I/O functions by changing one function for input and one for output. For the Keil libraries these are `getkey()` and `putchar()` and for the TASKING libraries these are `_ioread()` and `_iowrite()` respectively. The latter two get a parameter with a stream number passed, thus allowing you to support several independent I/O streams, each with its own mechanism.

3 COMPILER INVOCATION

3.1 MEMORY MODELS

The Keil compiler has three memory models, namely small, compact and large. The TASKING compiler has four models, the first three namely small, aux and large respectively are equivalent to the Keil models. The fourth TASKING model is reentrant.

The TASKING compiler always keeps the large virtual stack required for reentrancy in external memory, whereas the Keil compiler keeps it in the default memory determined by the used memory model.

Since the TASKING compiler has a reentrant model, it also has a library to go with it. The parameter passing to the functions included in this library use the reentrant convention, that is via the virtual stack if there are not enough registers.

3.2 LIBRARIES

Both compiler packages contain a library for each memory model. They both use the same naming conventions, thus the following shows the names of equivalent libraries.

Compiler Model	Keil Compiler	TASKING Compiler
Small	c51s.lib	c51s.lib
Compact/Auxpage	c51c.lib	c51a.lib
Large	c51l.lib	c51l.lib
Reentrant	–	c51r.lib

Table F–2: Libraries

The Keil compiler package also contains a floating point library for all three memory models. In contrast the TASKING package contains two floating point libraries, which can be used independent of the selected memory model. The most commonly used floating point library (called `float.lib`) keeps the floating point stack in external ram (XDATA), whereas the other one (called `floats.lib`) keeps it in internal ram (IDATA). The IDATA version contains only the most basic floating point operations. In general, the smaller one, with respect to data memory usage, is comparable with the small model version of the Keil package (`c51fps.lib`), and the other one is comparable with the large model version of the Keil package (`c51fpl.lib`).

3.3 CONTROLS OR PRAGMAS

For the Keil compiler all compiler controls can also be used as a pragma within the C source code. The TASKING compiler has some equivalent notation for most of these directives, either as a compiler command line option or a C source pragma or both, or as an assembler directive. The table below shows how to convert Keil directives to an equivalent TASKING notation.

Keil	TASKING	Comment
aregs noaregs		no translation
asm endasm	asm endasm	compiler pragma compiler pragma
code		assembly is always generated, use **–s** for mixing with source code
compact	–Ma	compiler option
debug nodebug	–g	compiler option
define	–D	compiler option
disable		no translation
eject	eject	assembler directive
interval		use assembly
intpromote nointpromote		handled automatically by compiler
intvector		use assembly
large	–Ml	compiler option

Keil	TASKING	Comment
listinclude	–li listinc	compiler option or compiler pragma
maxarg	–a arglist	compiler option or compiler pragma
noamake		no translation
nocond		no translation
noextend	–U_CC51 –Hcc51.h	compiler options
object noobject	object noobject	assembler directive assembler directive
objectextend		default behavior
optimize	–O optimize	compiler option or compiler pragma see following paragraph
order	–Ot	compiler option
pagelength	pagelength	assembler directive
pagewidth	pagewidth	assembler directive
preprint	–E	compiler option
print	print	assembler directive
regfile	–C	compiler option
registerbank	registerbank	assembler directive
regparms	–Or	compiler option
rom small rom compact rom large	–rs –rm –rl	compiler option compiler option compiler option
save	save	assembler directive
restore	restore	assembler directive
small	–Ms	compiler option
src	–o	compiler option
symbols		use linker map file

Table F–3: Controls, pragmas and options

3.4 COMPILER OPTIMIZATIONS

Both C compilers allow you to customize the compiler optimizations. The Keil compiler uses the `optimize` control or pragma and the TASKING compiler uses the **–O** option or the `optimize` pragma.

The TASKING compiler allows you to switch on or off most optimization mechanisms independent of other optimizations. This in contrast with the Keil compiler which only allows you to select a certain level of optimization except for optimization seperatly on speed or size.

The following table shows which TASKING optimizations options are comparable to the *extra* optimizations introduced for each Keil optimize level. Some of the Keil optimizations, however, cannot be switched off in the TASKING compiler, for example, constant folding (Keil level 0), data overlaying (Keil level 2), etc.

Keil Compiler	TASKING Compiler
size	F
speed	f
0	p
1	p
2	
3	h
4	w // and a #pragma see 4.4 *Pragmas* for switch statements
5	c, s
6	l

Table F–4: Optimization

The TASKING compiler offers you many more optimizations. Refer to the description of the **–O** option for a detailed description.

INDEX

INDEX

INDEX

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

INDEX

P

Q

R

S

T

U

V

W

X

INDEX