C166/ST10 v8.7

C Cross-Compiler
User's Manual

TASKING

Embedded software development from Altium

TASKING is a brand name of Altium Limited.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Macrovision Corporation.
Intel is a trademark of Intel Corporation.
Motorola is a registered trademark of Motorola, Inc.
MS-DOS and Windows are registered trademarks of Microsoft Corporation.
SUN is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.

All other trademarks are property of their respective owners.
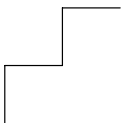
Data subject to alteration without notice.

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

CONTENTS

TABLE OF CONTENTS

TASKING

CONTENTS

CONTENTS

CONTENTS

CONTENTS

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the TASKING C166/ST10 C 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 C166/ST10.

2. Overview
   Provides an overview of the TASKING C166/ST10 toolchain and gives you some familiarity with the different parts of it and their relationship. A sample session explains how to build a C166/ST10 application from your C file.

3. Language Implementation
   Concentrates on the approach of the C166/ST10 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 control program and C compiler invocation, command line options and pragmas.

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

6. Libraries
   Contains the library functions supported by the compiler, and describes their interface and 'header' files.

• • • • • • • • •

7. Run-time Environment

> Describes the run-time environment for a C application. It deals with items like assembly language interfacing, C startup code and stack/heap size.

Appendices

A. MISRA C

> Supported and unsupported MISRA C rules.

B. Debug Environment

> Contains operation remarks when you want to use a debug environment such as CrossView Pro with evaluation boards, Kontron debugger, Hitex HiTOP or the pls *fast*-view66 debugger.

C. CPU Functional Problems

> Describes how the C166/ST10 toolchain can bypass some functional problems of the CPU.

D. User Stack Model

> Describes the special coding methods used in the libraries and C166/ST10 C compiler to support a special stack frame.

MANUAL STRUCTURE

RELATED PUBLICATIONS

- The C Programming Language (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ANSI X3.159-1989 standard [ANSI]
- C166/ST10 Cross-Assembler, Linker/Locator, Utilities User's Manual [TASKING, MA019-000-00-00]
- C166/ST10 C++ Compiler User's Manual [TASKING, MA019-012-00-00]
- C166/ST10 CrossView Pro Debugger User's Manual [TASKING, MA019-041-00-00]
- C16x User's Manuals [Infineon Technologies]
- ST10 User's Manuals [STMicroelectronics]
- ST10 Family Programming Manual [STMicroelectronics]
- XC16x/Super10 User's Manuals [Infineon Technologies / STMicroelectronics]

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

CHAPTER

1
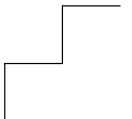
1.1   INTRODUCTION

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

The software for Windows has two faces: a graphical interface (Embedded Development Environment) and a command line interface. The Solaris software only has 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. Insert the TASKING CD-ROM into the CD-ROM drive.

2. If the installation program does not start automatically, browse to your CD-ROM drive and run the program **setup.exe**.

   The TASKING Setup dialog box appears.

3. Select a product and click on the **Install** button.

4. 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.

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

1.2.2   INSTALLATION FOR SOLARIS

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. Insert the TASKING CD-ROM into the CD-ROM drive and mount the
   CD-ROM on a directory, for example `/cdrom`. Make sure to use an ISO
   9660 file system with Rock Ridge extensions enabled. See the UNIX
   manual pages about **mount** for details.

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

       cd /cdrom

4. Run the installation script:

       sh install

   Follow the instructions that appear on your screen.

   First a question appears about where to install the software. The default
   answer is **/usr/local**.

   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 <=
   ```

INSTALLATION

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

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

5. 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 and Solaris.

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 Solaris), you can set *environment variables*.

You can set the following variables:

Environment Variable	Description
A166INC	With this variable you specify one or more additional directories in which the assembler **a166** looks for STDNAMES files.
C166INC	With this variable you specify one or more additional directories in which the C compiler **c166** looks for include files. The compiler first looks in these directories, then always looks in the default `include` directory relative to the installation directory.
CC166BIN	When this variable is set, the control program **cc166**, prepends the directory specified by this variable to the names of the tools invoked.
CC166OPT	With this variable you specify options and/or arguments to each invocation of the control program **cc166**. The control program processes these arguments before the command line arguments.

INSTALLATION

Environment Variable	Description
LINK166	With this variable you specify extra options and/or arguments to each invocation of the link stage of **l166**.
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 Solaris).
LOCATE166	With this variable you specify extra options and/or arguments to each invocation of the locate stage of **l166**.
M166INC	With this variable you specify one or more additional directories in which the macro preprocessor **m166** looks for include files.
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.
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 C166INC variable as an example.

See also section 4.4, *Include Files* in chapter *Compiler Use*.

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:

   ```
   C166INC
   ```

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

   ```
   c:\c166\include
   ```

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

Example for Solaris

Enter the following line (C-shell):

```
setenv C166INC /usr/local/c166/include
```

INSTALLATION

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 Altium for the type of license purchased.

You can run TASKING products with a node-locked license, with a
floating license or in trial mode. When you order a TASKING product
determine which type of license you need (Solaris 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.

Trial mode

When you use the product without a valid license, the tools will run in
trial mode. This means you can use the toolset 15 days with full
functionality. When you run the toolset in trial mode, each tool will report
this. When you use a license that does not cover the full toolset, the tools
that are not covered by the license will run in trial mode.

When after you installed the license file, the tools that are covered by the
license still report that they are running in trial mode, this means that there
is a license error. If you want to force the termination of the trial mode to
get the FLEXlm error message you can set the environment variable
`FORCE_NO_TRIAL` to "yes".

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.

Solaris

1. If you need a floating license on Solaris, 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.

INSTALLATION

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.

For 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.

For Solaris, 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 Solaris.

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.

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.

INSTALLATION

1.4.4   MODIFYING THE LICENSE FILE LOCATION

The default location of the license file for Windows is:

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

For Solaris 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 Solaris ':'):

Example Windows:

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

Example Solaris:

```
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
Windows	**licadmin** (License Administrator, or use **lmhostid**)	0060084dfbe9
Solaris	**hostid**	170a3472

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.
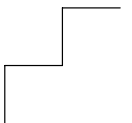
Platform	Method
Windows	**licadmin** or: Go to the Control Panel, open "System". In the "Computer Name" tab look for "Full computer name".
Solaris	**hostname**

Table 1-3: Determine the host name

INSTALLATION

CHAPTER 2

OVERVIEW

TASKING

CHAPTER

2

2.1 INTRODUCTION TO C C166/ST10 CROSS-COMPILER

This manual provides a functional description of the TASKING C C166/ST10 Cross-Compiler. This manual uses **c166** (the name of the binary) as the shorthand notation for 'TASKING C C166/ST10 Cross-Compiler'.

TASKING offers a complete toolchain for the Infineon C166 and STMicroelectronics ST10 microcontroller families and their derivatives. These derivatives can be based on C16x/ST10x extended architectures (16M memory, 24 bit addresses) and XC16x/Super10 extended architectures. This manual uses 'C166/ST10' as the shorthand notation for these microcontroller families. The toolchain contains a C++ compiler, a C compiler, a control program, a macro preprocessor, an assembler, a linker/locator, a library manager, a program builder, a disassembler, a debugger and output format utilities.

The **c166** is not a general C compiler adapted for use with the C166/ST10 architecture, but instead it is dedicated to the microcontroller architecture of the C166/ST10 architecture. This means that you can access all special features of the C166/ST10 architecture in C: 16K page architecture (with full pointer support), bit-addressable memory, (extended) special function registers (I/O ports), interrupt support, scalable vector tables, (local) register banks and a number of built-in (intrinsic) functions to utilize special C166/ST10 architecture 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.

The **c166** generates assembly source code using the Infineon assembly language specification, and must be assembled with the TASKING C166/ST10 Cross-Assembler. This manual uses **a166** as the shorthand notation for 'TASKING C166/ST10 Cross-Assembler'.

The object file generated by **a166** can be linked with other objects and libraries using the TASKING **l166** linker/locator. This manual uses **l166** as the shorthand notation for 'TASKING **l166** linker/locator'. With the link stage of **l166** you can link objects and libraries to one object. You can locate assembler objects, linked objects and libraries to a complete application by using the locate stage of **l166**.

• • • • • • • • •

The C166/ST10 toolchain also accepts C++ source files. C++ source files or sources using C++ language features must be preprocessed by **cp166**. The output generated by **cp166** is C166/ST10 C, which can be translated with the C compiler **c166**.

The C++ compiler is not part of the C compiler package. You can order it separately from TASKING. The C++ compiler package includes the C compiler as well.

With the TASKING **cc166** control program you can invoke the various components of the C166/ST10 toolchain with one call. This manual uses **cc166** as the shorthand notation for 'TASKING **cc166** control program'.

You can debug the software written in C, C++ and/or assembly with the TASKING CrossView Pro high-level language debugger. This manual uses XVW166 as the shorthand notation for 'TASKING CrossView Pro high-level language debugger'. A list of supported platforms and emulators is available from TASKING.

You can also use other debugging environments supporting the IEEE-695 format (e.g. Kontron, Hitex, Krohn & Stiller, Lauterbach, etc.).

Target Processors:

All <u>C16x/ST10</u> derivatives (such as C167, ST10x172). This is the default.
All <u>ST10</u> derivatives with MAC support is enabled with the **'-xd'** option.
All <u>C166S v1.0</u> derivatives support is enabled with the **'-x1'** option.
All <u>XC16x/Super10</u> derivatives support is enabled with the **'-x2'** option.
All <u>enhanced Super10</u> derivatives support is enabled with the '**-x22**' option.

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*.

OVERVIEW

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:

> This phase performs target processor independent optimizations by
> transforming the intermediate code. The next section discusses
> 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. Chapter 3, *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 C166/ST10 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). It also performs pipeline optimizations, replacing NOP instructions with other instructions which do not interfere with the pipeline effects of the processor. Another task of the peephole optimizer is to convert JMPR instructions to JMPA 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 **a166**. The generated assembly does not contain any macros.

The data flow analysis (DFA) peephole optimizer phase:

> This phase uses data flow analysis (DFA) to perform optimizations on the assembly code. This optimizer runs after the normal peephole optimizer described above. The optimizer has function scope and solves a number of DFA problems. With the analysis results, optimizations can be performed without being hampered by flow changing instructions.

The instruction reordering phase:

> This phase is only enabled for the ext2 architectures. It tries to reorder the instructions in order to keep the pipeline from stalling as much as possible. During this phase no instructions will be added or removed.

All phases (of both frontend and backend) are combined into one program: **c166**. 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 following optimizations are performed 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.

OVERVIEW

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. These situations frequently occur with nested control structures. This optimization does not save code, but speeds up execution.

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.

Register coloring

Optimize register allocation within a C function. The compiler tries to keep as much local variables as possible in registers.

Constant/value propagation

A reference to a variable with a 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.

Sharing of string literals and floating point constants

The ANSI X3.159-1989 standard permits string literals to be put in ROM memory. Strings in ROM cannot be modified, so 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.

Common Tail Merging

Common pieces of code at the end of case labels and if-else constructions are replaced by a jump to single instance of the shared code. This will reduce code size.

OVERVIEW

2.3   PROGRAM DEVELOPMENT FLOW

If you want to build a C‑166 application you need to invoke the following programs:

- The C compiler (**c166**), which generates an assembly source file from the file with suffix `.c`. The suffix of this file is `.src`, which is the default for **a166**. 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 by means of the **–s** option. High level language debugging information can be generated with the **–g** option. You should not use the **–g** option, when inspecting the generated assembly source code, because it contains a lot of 'unreadable' high level language debug directives. **c166** makes only one pass on every file. This pass checks the syntax, generates the code and performs a code optimization.

- The **a166** cross‑assembler 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 **l166** link stage which links the generated relocatable object files and C‑libraries. The result is a relocatable link file with suffix `.lno`. A linker task map file with suffix `.lnl` is available after this stage.

- The **l166** locate stage which locates the generated relocatable object files (from assembler or link stage). The result is a loadable file with suffix `.out`. A full application map file with suffix `.map` is available after this stage.

- The **ieee166** program which formats an `a.out` type file into a CrossView Pro load file.

The next figure explains the relationship between the different parts of the TASKING C166/ST10 toolchain:

• • • • • • • • •
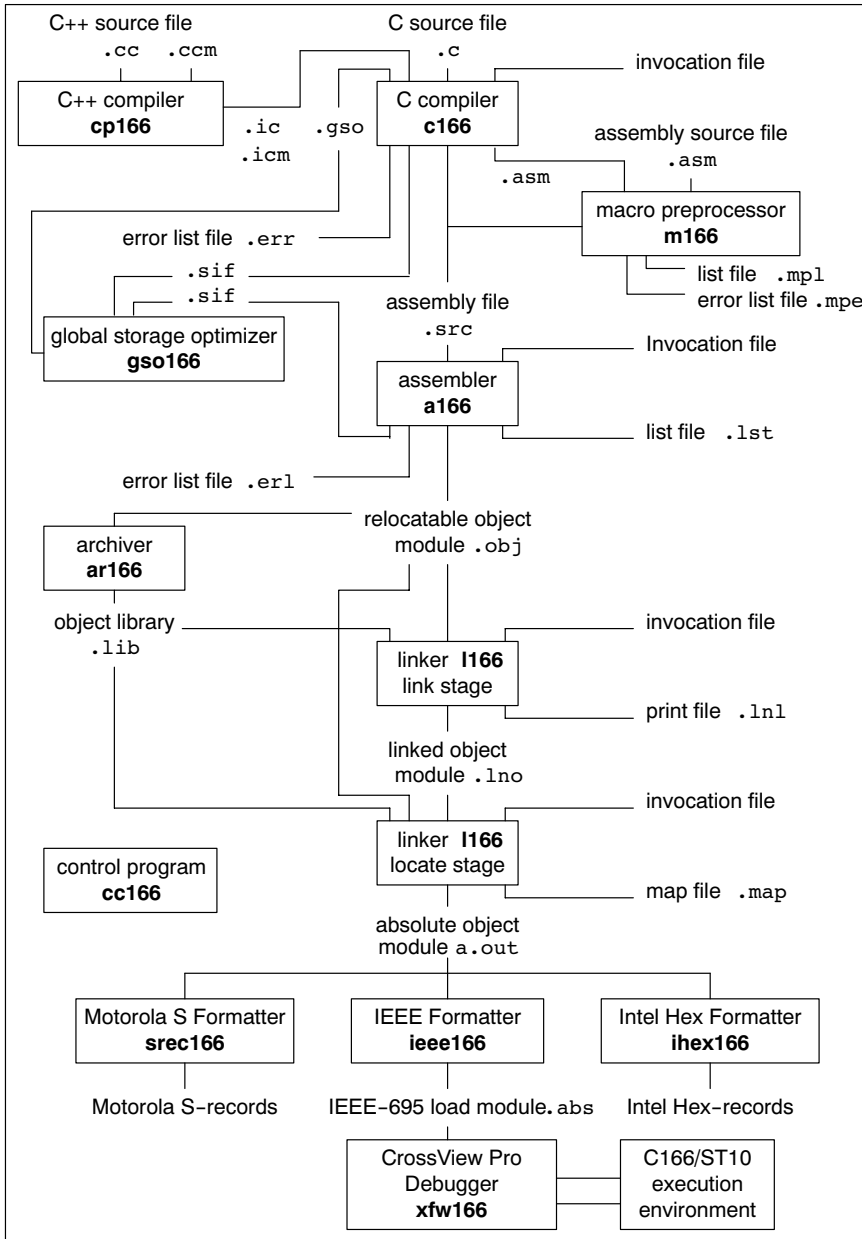
Figure 2-1: C166/ST10 development flow

You can use the control program **cc166** to build an absolute loadable file starting with an input file of any stage. C++ source programs are compiled by the C++ compiler. With a C source file as input, **cc166** calls **c166**, **a166** and **l166** with the appropriate command line arguments.

It is advised to use **cc166** when you compile C++ source programs because of the complex nature of C++ compilation.

The global storage optimizer **gso166** is a program to optimize allocation of objects in memory spaces.

The macro preprocessor **m166** is a program to preprocess assembly files (suffix `.asm`).

The **ihex166** program formats the `a.out` file into an Intel Hex format file. You can load this output file into an EPROM programmer.

The **srec166** program formats the `a.out` file into a Motorola S Format for EPROM programmers.

The **ar166** program is a librarian facility. You can use this program to create and maintain object libraries.

A utility to disassemble absolute object files and relocatable object files is **d166**.

A utility to display the contents of an object file is **dmp166**.

The **mk166** program builder uses a set of dependency rules in a 'makefile' to build only the parts of an application which are out of date

For a full description of all available utilities, see chapter 12 *Utilities* in the *C166/ST10 Cross-Assembler, Linker/Locator, Utilities User's Manual*.

The name of the C166/ST10 CrossView Pro Debugger is **xfw166**. For more information check the *C166/ST10 CrossView Pro Debugger User's Manual*. This manual uses **xvw166** as the general executable name.

File extensions

The following table lists the file types used by the C166/ST10 toolchain.

Extension	Description
Source files	
.cc, .cxx, .cpp	C++ source file, input for the C++ compiler, compiled to .ic
.ccm	C++ source file containing intrinsics, input for the C++ compiler, compiled to .icm
.c	C source file, input for the C compiler
.cmp	C source file containing intrinsics, input for the C compiler
.asm	Assembler source file, hand coded, or generated by C compiler from .cmp or .icm
Generated source files	
.ic	C source file, generated by the C++ compiler, input for the C compiler
.icm	C source file containing intrinsics, generated by the C++ compiler, input for the C compiler
.src	Assembler source file, generated by the C compiler
.sif	Source information file for the global storage optimizer
.gso	Global storage optimizer file
Object files	
.obj	IEEE–695 relocatable object file, generated by the assembler
.lno	Linked object module
.lib	Object library file
.out	Absolute locator output file
.abs	IEEE–695 absolute object file
.hex	Intel Hex absolute object file
List files	
.mpl	Macro proprocessor list file
.lst	Assembler list file
.lnl	Linker map file
.map	Locator map file
.mcr	MISRA C report file

Extension	Description
Error list files	
.err	Compiler error messages file
.mpe	Macro preprocessor error messages file
.erl	Assembler error messages file
.elk	Linker 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

OVERVIEW

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.

Project Options   Compile   Build   Rebuild   Debug   On–line Manuals

Document Windows
Used to view and edit files.

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

Output Window
Contains several tabs to display and manipulate results of EDE operations. For example, to view the results of builds or compiles.
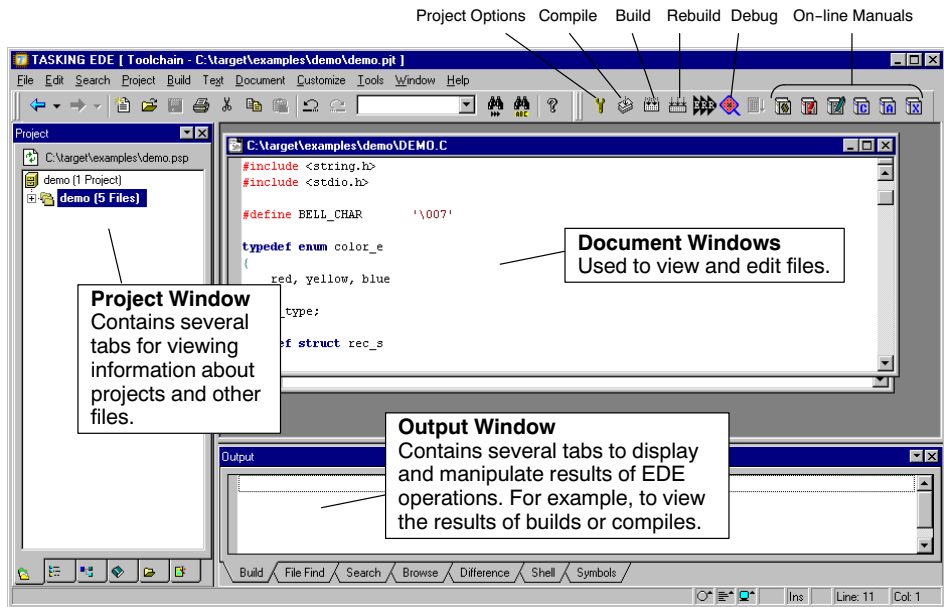
Figure 2-3: EDE desktop

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.*

OVERVIEW

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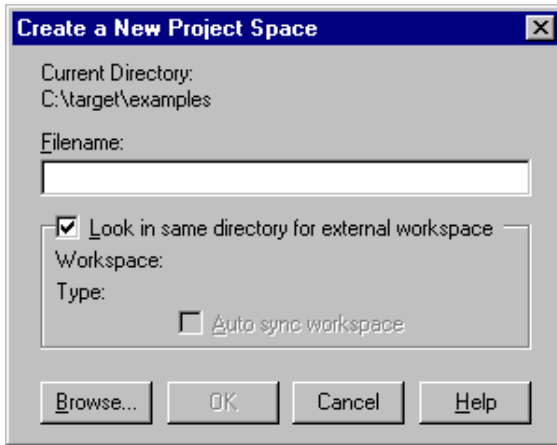
2. In the the **Filename** field, enter a name for your project space (for example `MyProjects`). Click the **Browse** button to select a directory first and enter a filename.

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.
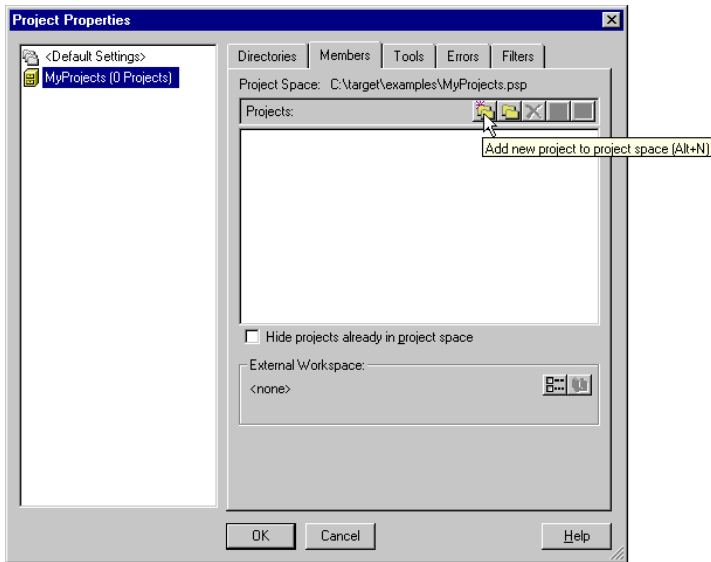
Add a new project to the project space

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.

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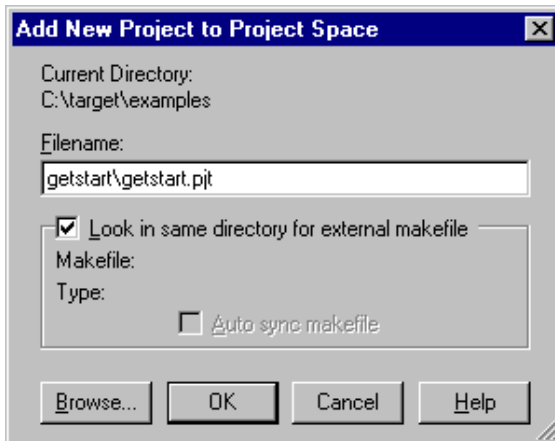
5. Give your project a name, for example `getstart\getstart.pjt` (a directory name to hold your project files is optional) and click **OK**.

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.*
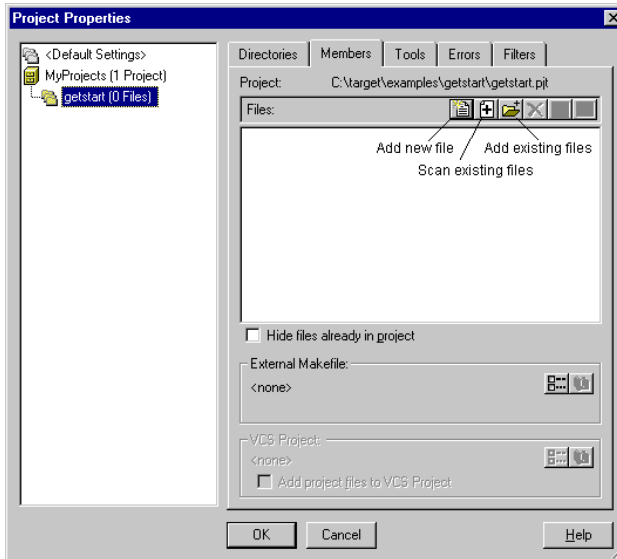
Add new files to the project

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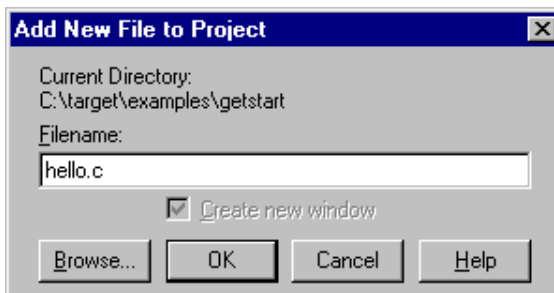
7. Enter a new filename (for example `hello.c`) and click **OK**.

    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 **Application** entry and select **Processor**.
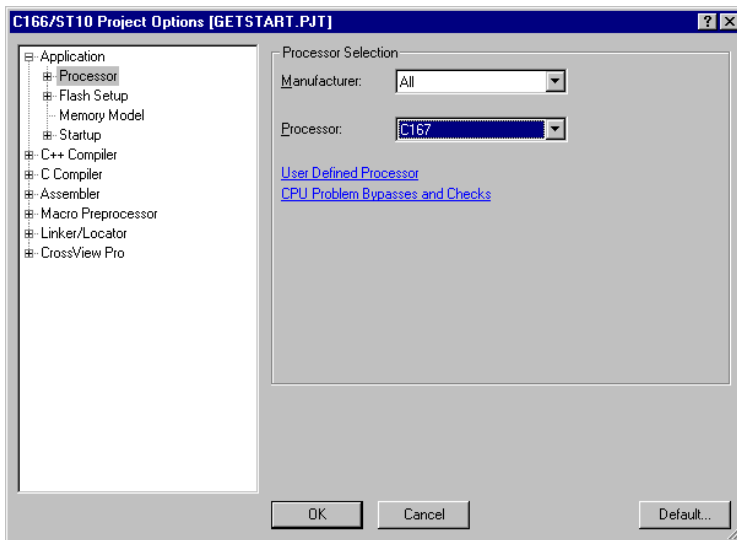
3. Optionally select a **Manufacturer** to narrow the list of processors.

4. In the **Processor** list select your target processor (for example, **C167**).

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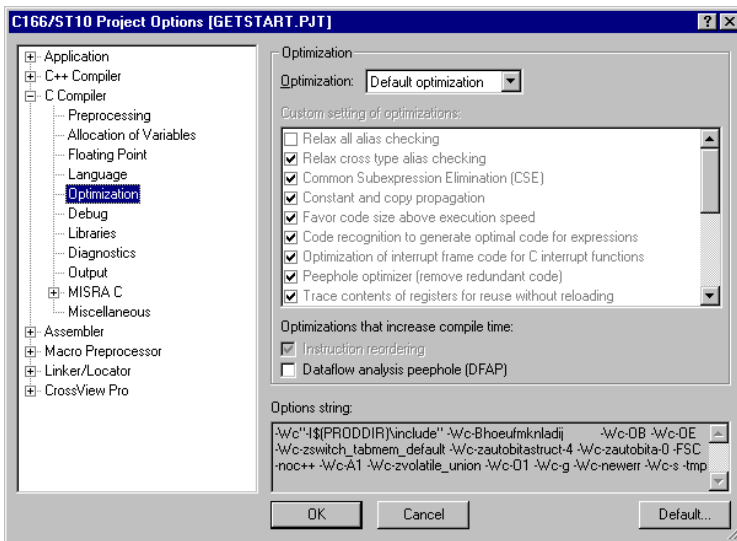
3. For each page make your changes. If you have made all changes click **OK**.

   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 (C++ Compiler, Assembler, Macro Preprocessor, Linker/Locator, CrossView Pro) 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* **cc166**.

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`:

    cc166 –g –ieee –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 THE CONTROL PROGRAM

In order to debug your programs, you will have to compile, assemble, link and locate them for debugging using the TASKING C166/ST10 tools. You can do this with one call to the control program.

To use the control program on the `sieve` demo program in the subdirectory `sieve` in the `examples` subdirectory of the C166/ST10 product tree follow the steps below. This procedure is outlined as a guide for you to build your own executables for debugging.

OVERVIEW

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

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 control program **cc166**:

    cc166 –g –ieee –o sieve.abs sieve.c

    The **–g** option instructs the compiler to generate symbolic debugging information. If you want to debug your program with the CrossView Pro high level language debugger, this option must be on.

    The **–ieee** option specifies that the output file must be formatted in the IEEE Std. 695 format. The **–o sieve.abs** option specifies the output filename to be `sieve.abs`. The result of the command are the files `sieve.abs` which can be loaded and executed by CrossView Pro and `sieve.map` containing the locate map of the application.

    You can specify the **–DMEASURE_TIME** option if you want to build the sieve benchmark program for time measurement. Note that this is done in the `makefile` which can be processed by **mk166**.

    Now you have created all the files necessary for debugging with CrossView Pro with one call to the control program.

    If you want to see how the control program calls the compiler, assembler, link stage, locate stage and formatter, you can use the **–v** option or **–v0** option. The **–v0** option only displays the invocations without executing them. The **–v** option also executes them:

    cc166 –g –ieee –o sieve.abs sieve.c –v0

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

```
+ c166 sieve.c -o /tmp/cc5882c.src -e -g
+ a166 /tmp/cc5882c.src TO sieve.obj NOPR
+ l166 LNK TO /tmp/cc5882d.lno sieve.obj ext/c166s.lib ext/fp166s.lib
         ext/rt166s.lib
+ l166 LOC TO /tmp/cc5882e.out /tmp/cc5882d.lno NOPR
+ ieee166 /tmp/cc5882e.out sieve.abs
```

The **–e** option specifies to remove the output file if compiler errors occur. The **NOPR** control suppresses the assembler list file generation. The **TO** control has the same function as the **–o** option of the compiler, and specifies the output filename.

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

```
cc166 –g –ieee –o sieve.abs sieve.c –v0 –tmp
```

This command produces the following output:

```
+ c166 sieve.c -o sieve.src -e -g
+ a166 sieve.src TO sieve.obj NOPR
+ l166 LNK TO sieve.lno sieve.obj ext/c166s.lib ext/fp166s.lib
        ext/rt166s.lib
+ l166 LOC TO sieve.out sieve.lno
+ ieee166 sieve.out sieve.abs
```

As you can see, if you use the **–tmp** option, the assembly source files and linker output file will be created in your current directory also.

Of course, you will get the same result if you invoke the tools separately using the same calling scheme as the control program.

As you can see, the control program automatically calls each tool with the correct options and controls.

2.10.2 USING THE SEPARATE PROGRAMS

If you want to call each tool separately instead of using the control program you can issue the following commands (steps 3-7 replace step 3 of the previous section).

3. Compile the module:

```
c166 –s –g –t sieve.c
```

The **–s** option puts the C source text as comments into the output assembly source file `sieve.src`. The other options are the same as explained by the invocation of the control program.

4. Assemble the module:

```
a166 sieve
```

OVERVIEW

The suffix `.src` is default and may therefore be omitted. The assembler produces a relocatable object file called `sieve.obj` and a list file called `sieve.lst`.

If you want to build a complete C166/ST10 executable application, the module containing the C function `main()` is treated like a reset task and therefore must be linked with the C startup code. When the Task Concept is followed, all tasks should be linked with a library, that contains, among run-time routines, functions such as `printf()`. When the Flat Interrupt Concept is followed the C startup code and the library is linked in the locate stage and the link stage is skipped. In this example we are using the Task Concept.

The C startup code is delivered in each run-time library for the memory model of the library and in assembly source code, because this file usually must be adapted to the target environment. The library is delivered for all memory models supported. In this case, we are using the small model, because this is the default memory model of **c166**. See the next chapter for detailed information on memory models.

The libraries are organized in two basic library sets: one set for the C16x/ST10 architecture (subdirectory `ext`) and one set for the XC16x/Super10 architectures (subdirectory `ext2`).

These two basic library sets are additionally organized in two variants: one standard variant and one variant with all silicon bug workarounds enabled. The subdirectories for this last variant are followed by the character 'p' (subdirectories `extp` and `ext2p`).

All four library sets are also available for the User Stack Model. All subdirectories for this extra variant are preceded with the character 'u'.

It depends on the hardware environment you are using, which library set must be used. By default the compiler assumes the C16x/ST10 architecture without any silicon bug workarounds enabled. Therefore, the library set in the subdirectory `ext` is used.

5. Link the module by typing:

   PC:

   ```
   l166 link sieve.obj ext\c166s.lib ext\rt166s.lib to
   sieve.lno
   ```

UNIX:

```
l166 link sieve.obj ext/c166s.lib ext/rt166s.lib to
sieve.lno
```

By default the linker searches the `lib` directory for libraries. This way it finds the `c166s.lib` and `rt166s.lib` libraries. The `cstart.obj` C startup code is extracted from the `rt166s.lib` library because the compiler generates a reference to this module when the `main()` function is defined.

The result of this command is the linked task object module `sieve.lno`. When you use the **PRINT** control the file `sieve.lnl` is created, containing information about the linking stage: memory map, symbol table, register map. However, this is slowing down the process of linking and therefore turned off by default.

6. Locate the module by typing:

```
l166 locate sieve to sieve.out nocc
```

The result of this command is the absolute output file `sieve.out` and the file `sieve.map` containing the locate map of the application. The **nocc** control disables the checking on definition of class ranges, used to locate all parts of the application in user defined memory ranges.

In order to load this application into the CrossView Pro debugger, the output file must be formatted into IEEE Std. 695 format.

7. Format the output file by typing:

```
ieee166 sieve.out sieve.abs
```

The file `sieve.abs` can be loaded and executed by CrossView Pro.

OVERVIEW

2.10.3 USING A MAKEFILE

The `examples` directory contains several subdirectories with example programs. Each subdirectory contains a `makefile` which can be processed by **mk166** to build the example.

The `examples` directory also contains a `makefile` for building all examples. For building all examples, add the `bin` directory of the installed product to the search path and type:

```
mk166
```

For building one example program, make the directory containing the example the current working directory. Build the example by typing:

```
mk166
```

When the example has already been built before, only the parts which are out of date are rebuilt.

For more information see also the `readme.txt` files in the subdirectories of the examples.

To see which commands are invoked by **mk166** without actually executing them, type:

```
mk166 -n
```

When you want to re-translate the examples with other settings you should first clean up the results of a previous translation. This can be done by:

```
mk166 clean
```

You can also use this when you just want to clean up the example directories.

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.

 See the *CrossView Pro Debugger User's Manual* for more information.

OVERVIEW

2.12 USING DAvE PROJECTS WITH EDE

Infineon Technologies' DAvE 2.x is fully supported by means of its generated project information file (`*.dpt`). This means that you can easily import projects created with DAvE in the C166/ST10 EDE. The memory model, the CPU and startup register settings (in the **Project | Project Options** dialog) will reflect the settings you made in DAvE. In addition all files created by DAvE will be added automatically to your own EDE project when you press the 'Refresh DAvE imported project' button (this button only appears when your EDE project contains a DAvE generated project information file).

How to add DAvE projects to EDE

Create your DAvE project (for example "`my_project.dav`") and generate code for the TASKING C166/ST10 products. Now follow these steps:

1. Create a new (or open your existing) C166/ST10 EDE project (for example "`ede_project.pjt`"). For more information on how to do this, see section 2.7, *Create a New Project Space with a Project*.

2. Add the DAvE generated project information file ("`my_project.dpt`") to the project: in the Project Properties dialog click on the **Add existing files to project** button and select the DAvE project.

   *This file appears in the **Other Files** category of your project.*

3. Add the C startup code `start.asm` to your project: from the **Project** menu, select **Project Options...**, expand the **Application** entry and select **Startup**, enable the check box **Generate system startup code and add it to project** and specify the name `start.asm` in the **Startup code file name** field.

4. Click the **Refresh DAvE imported project** button in the EDE toolbar.

   The EDE project is now fully setup to build the application you have created using DAvE.

The C166/ST10 EDE only reflects those (E)SFR register settings which must be configured when booting the CPU before the execution of the EINIT (end of initialization) instruction. These registers are configured in the C startup code. All other registers are configured from the C code which is generated by DAvE.

Every update to the DAvE project will automatically be imported in your EDE project when you press the '**Refresh DAvE imported project**' button. This will override any settings with respect to the memory model, the CPU and startup register settings you have made from the EDE (in the **Project | Project Options** dialog), because these settings are already defined by you from within DAvE. All other C166/ST10 EDE settings keep your configured values. Changes you make manually to your EDE project settings and the source code which is generated by DAvE, cannot be imported back into your DAvE project. Therefore, this should only be done if you plan not to use DAvE anymore for making changes to your project files and settings.

OVERVIEW

CHAPTER

3

LANGUAGE IMPLEMENTATION

TASKING

3.1   INTRODUCTION

The TASKING C C166/ST10 cross-compiler offers a new approach to high-level language programming for the C166/ST10 family. It conforms to the ANSI standard, but allows the user to control the I/O registers, bit memory, interrupts and data page architecture of the C166/ST10 in C. This chapter describes the language implementation in relation to the C166/ST10 architecture.

The extensions to the C language in **c166** are:

_bit

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

_bitword

You can declare word variables in the bit-addressable area as fp. You can access individual bits using the intrinsic functions `_getbit()` and `_putbit()`.

_sfrbit / _esfrbit

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

_sfr / _esfr

Data types for the declaration of Special Function Registers.

_xsfr

Data type for the declaration of Special Function Registers not residing in SFR memory but do reside in internal RAM. An example of these SFRs are PEC source and destination pointers. The compiler will use a 'mem' addressing mode for this data type whereas for an object of type `_sfr` a 'reg' or 'mem' addressing mode may be used.

These SFRs are not bitaddressable.

_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 `_bitword` or bitaddressable `_sfr` variable.

_inline

Used for defining inline functions.

_usm / _nousm

With these function qualifiers you can force that a function is called using the user stack model calling convention or using the generic CALL/RET calling convention.

_bita

You can tell the compiler that a struct must be located in bitaddressable memory by using the `_bita` memory qualifier.

memory-specific pointers

c166 allows you to define pointers which point to a specific target memory. These types of pointers are very efficient and require only 2 or 4 bytes memory space.

special 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 C166/ST10 (`_near`, `_xnear`, `_far`, `_huge`, `_shuge`, `_system`, `_iram`).

interrupt functions

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

intrinsic functions

A number of pre-declared functions can be used to generate inline assembly code at the location of the intrinsic (built-in) function call. This avoids the overhead which is normally used to do parameter passing and context saving before executing the called function.

LANGUAGE

3.2 ACCESSING MEMORY

The C166/ST10 allows to access memory up to 16 MB using a 24-bit address. The processor does not use a linear addressing method (as the Motorola 68000 family), but uses a segmented approach of its memory (as the Intel 8086 family). Therefore, the difference in address range is only visible in the amount of bits in the segment/page registers.

The approach of data memory differs with the approach of code memory. Code memory is accessed in segments of 64K using a 16-bit offset and an 8-bit segment number. Because there is no translation done on this 8-bit segment number, code memory access is 'almost' linear. However, data memory is accessed within 16 KB pages. The 16-bit address is translated into a 24-bit address via one of four data page pointers, specified with bit 14 and 15. So, the 24-bit address is made out of the 14-bit page offset and the 10-bit contents of the selected DPP.

c166 has two methods of gaining greater control over how your program uses memory. These methods can be used together. First you can specify the 'memory model' for the program. The compiler allows you to choose from a number of different approaches. In section 3.2.1 *Memory Models* more detailed information is present. Second, you can use one of the keywords _near, _xnear, _system, _iram, _far, _huge and _shuge in your program. Note that although these keywords are also used by other C compilers (for the 8086 family), they are not part of the standard C language. C is meant as a portable language.

In practice the majority of the C code of a complete application will be standard C (without using any language extension). This part of the application can be compiled without any modification, using the memory model which fits best to the requirements of the system (code size, amount of external RAM etc.). Therefore, **c166** has a number of features optimizing data access on standard C in all memory models. Note that a special section is present called 3.2.1.7, *Efficiency in Large Data Models*.

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

- I/O, using the (extended) special function registers
- high execution speed needed
- high code density needed
- access to non-default memory required (e.g. far/huge/shuge data)
- bit type needed

- C interrupt functions

3.2.1   MEMORY MODELS

c166 supports five memory models: tiny, small, medium, large, huge. You can select one of these models with the **–M** option. If you do not specify a memory model on the command line, **c166** uses the small memory model by default. The memory models with their characteristics are represented in the following table:

Model	DPP usage	$SEGMENTED control	CPU segmented mode	normal data size	code size	far/ huge/ shuge data allowed	near data allowed
tiny	linear	no	no	<64K	<64K	no	n.a.
small	linear	no	yes	<64K	>64K	yes	n.a.
medium	paged	yes	yes	>64K	<64K	yes	yes
large	paged	yes	yes	>64K	>64K	yes	yes
huge	paged	yes	yes	>64K	>64K	yes	yes
n.a. = not applicable							

Table 3-1: Memory models

The memory models can be described as follows:

3.2.1.1 TINY MEMORY MODEL

This memory model is the only model where the processor does <u>not</u> run in segmented mode, limiting the sum of code and data space to 64K. The DPP registers always contain their startup values thus allowing linear 64K access of data. This results in relatively high code density and execution speed. On interrupt the C166/ST10 does not have to save the CS register and an extra port (Port 4) is available, because address lines A16 - A23 are <u>not</u> used. The usage of the **_far**, **_huge** and **_shuge** keywords is not allowed. The tiny memory model is meant for very small (even single-chip) applications.

Map example
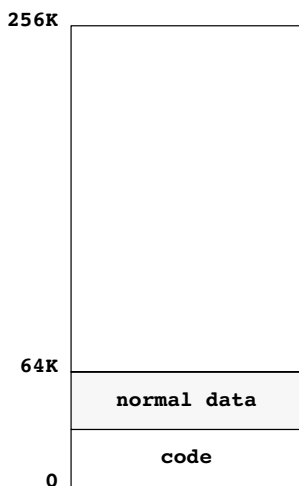
Figure 3-1: Tiny memory map example

Item	Usage	Comments
CPU	non–segmented	only model which runs non–segmented.
code	< 64K	limited to first segment of 64K.
normal data	< 64K	limited to first segment of 64K. Thus: (code + normal data) < 64K.
far data	not allowed	–
huge data	not allowed	–
shuge data	not allowed	–

Table 3-2: Tiny memory model

3.2.1.2 SMALL MEMORY MODEL

The small memory model is the most used memory model. It allows you to have a total code size up to 16M, up to 64K of fast accessible 'normal user data' in three different memory configurations and the possibility to access far/huge data, if more than 64K of data is needed.

The compiler does not assume the CSP register to contain something valid. Each call results in a far inter-segment code access, unless the _near keyword is used explicitly in the function prototype. We therefore recommend using the _near keyword with static functions when using the small or large model, since static functions are always in the same code section as their caller functions. This model allows code access in all segments up to 16M.

The small memory model supports 64K of 'normal user data' via fixed DPP values, specified at locate time. This results in high code density and execution speed. Note that the ROM data of an application (e.g. strings, floating point constants, jump tables, etc.) must also be allocated in this area of 64K of 'normal user data'. There are three memory configurations possible for this 64K of 'normal user data':

I (default)

The four DPP registers are assumed to contain their system startup value (0-3), providing one linear data area of 64K in the first segment (0-0FFFFh).

II Addresses Linear

DPP3 contains page number 3, allowing access to SYSTEM (extended) SFR registers and bitaddressable memory. DPP0 - DPP2 provide a linear data area of 48K anywhere in memory. You must specify the 'base-page-number' of this area at locate time via the **ADDRESSES**(**LINEAR**(*address*)) locator control.

III SND

DPP3 contains page number 3, allowing access to SYSTEM (extended) SFR registers and bitaddressable memory. DPP0, DPP1 and DPP2 contain the page number of a data area of 16K anywhere in memory. These page numbers are specified at locate time via the **SND** locator control. When you use this configuration, the size of a single 'normal data' object is limited to 16K.

In variant I and II, the paging principle is not really used, so the size of a single 'normal data' object (e.g. array) can be greater than 16K (one page).

If you use the small memory model (default of **c166**), the compiler uses the section type 'LDAT' for normal user data. This means that a non-paged section (unless SND is used of course) must be allocated by the locator in either:

I   first segment of 64K (default)
II   linear area of 48K specified with ADDRESSES LINEAR
        or in page 3
III   one of the three possible areas of 16K specified with SND
        or in page 3

If you need more than 64K of data (or if you need a huge data object), you can use the _far/_huge keywords in the declaration of these variables.

Small model memory map examples

Example **I**   Default

Example **II**   Using locate control:

```
AD   LINEAR( page 8 )
```
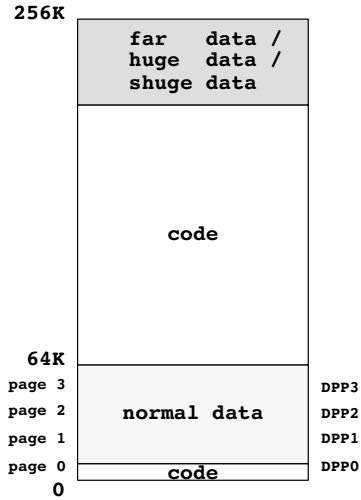
Example **III** Using locate control:
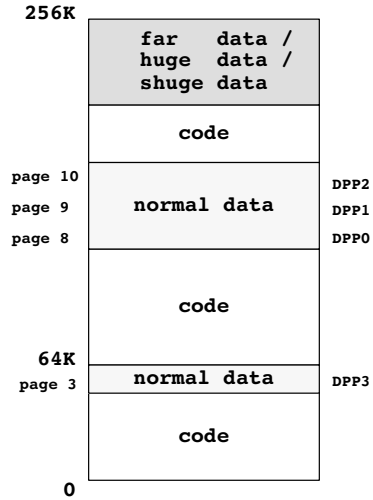
```
SND( DPP0(10), DPP1(12), DPP2(7)  )
```

'normal data' sections can contain both RAM data and ROM data.

Map example **I**

Map example **II**

Map example **III**
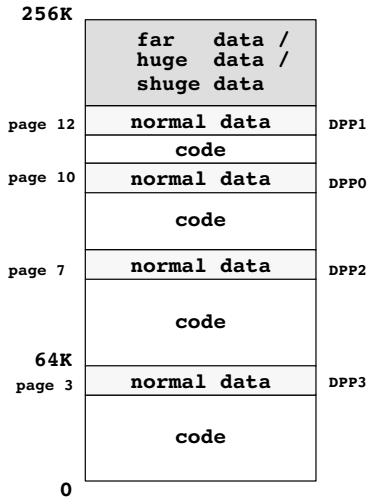
Figure 3-2: Small memory map examples

Item	Usage	Comments
CPU	segmented	–
code	>64K	allows code anywhere in 256K/16M.
normal data	< 64K	64Kb of fast accessible user data using direct MEM addressing mode. Except for map III (SND control), the size of a single user data object is **not** limited to 16K (16 bit address arithmetic). Also contains ROM data.
far data	allowed (optional)	supports far data (paged) access anywhere in 256K/16M. The size of a single far object is limited to 16K. Far data access is less fast than normal data access.
huge data	allowed (optional)	supports huge data access anywhere in 256K/16M. The size of a single huge object is **not** limited to 16K (32 bit address arithmetic). Huge data access is less fast than far data access. Size of one struct < 64K. Array of struct/any type > 64K
shuge data	allowed	supports shuge data access anywhere in 256K/16M. The size of a single shuge object is limited to 64K (16 bit address arithmetic). Shuge data access is as fast as huge data, but arithmetic on shuge addresses is faster.

Table 3-3: Small memory model

ROM data (e.g. strings, floating point constants, jump tables, etc.) is also present in LDAT sections and thus needs some space in the 64K of 'normal user data'. We recommend using page 3 for (external) ROM, allowing this ROM data (and code sections) to be allocated in this page and yet use DPP3 for SYSTEM (SFR) access. This means that the other three pages can be used for (external) RAM.

In the small model far/huge/shuge data access causes the compiler to emit code which, temporarily, overrules DPP0 with the page number of the far data. The DPP0 register is restored afterwards. DPP2 is sometimes used for far/near copy actions. During a task switch (interrupt) DPP0 and DPP2 are preserved and the correct page number is assigned to these DPP registers before activating the C code of this task, because a far access might be interrupted. The compiler also uses the special prefix instructions, which are treated by the processor as a prefix for a number of so-called 'atomic instructions': thus uninterruptable.

Far/huge/shuge data access produces extra code and results into slow execution. Therefore accessing far/huge/shuge data must be an exception within the application. The majority of the execution time of the application should be dealing with normal data, otherwise it is better to use the large model, allowing more efficient usage of far/huge/shuge data.

Far data is allocated in 'PDAT' sections, telling the assembler/linker/locator that a 'paged section' (must be checked to be in-page) is needed, which can be anywhere in memory. Huge data is allocated in 'HDAT' sections, specifying that a 'non-paged' (no checking for 16K) is needed, which can be anywhere in memory. Shuge data is allocated in 'SDAT' sections, which have the same properties as HDAT sections. The difference is that address calculations on shuge data is done in 16 bit rather than in 32 bit as with huge data. This implies that no shuge object can exceed 64K.

The following scheme is used for the data section types:

Section type	NON–SEGMENTED DATA (tiny/small)		SEGMENTED DATA (medium/large/huge)	
	meaning	location	meaning	location
DATA	paged (<16K)	1st segment: <64K	paged (16K)	anywhere
LDAT	linear(<64K)	tiny: 1st segment: <64K small: method I, II or III	–	–
PDAT	paged (<16K)	anywhere	–	–
HDAT	non–paged	anywhere	non–paged	anywhere
SDAT	–	–	non–paged	anywhere

Table 3-4: Small memory data section types

LDAT and PDAT section types are <u>not</u> allowed in segmented data mode. The only section type allowed in a DGROUP is the DATA type (not HDAT).

3.2.1.3 MEDIUM MEMORY MODEL

The compiler assumes that the CSP register contains the initial value of 0, which allows code access in the first 64K segment. The four DPP registers do <u>not</u> contain the system startup values. The DPP registers are used to access the 16M of data in 16K pages. Because the paging principle is used with 14 bit address arithmetic, data objects (e.g. arrays) cannot be greater than 16K (one page), unless the `_huge` or `_shuge` keyword is used. The `_huge` keyword tells the compiler to generate 24 bit address arithmetic. The `_shuge` keyword tells the compiler to generate 16 bit address arithmetic. Because paging is used, the processor must run in segmented mode. Exceptional access to code beyond 64K is possible declaring a huge function. However, it is not allowed for such a huge function to call any standard C (or run-time) library function, or any other 'near function' in the first segment. In section 3.2.1.7, *Efficiency in Large Data Models*, some details are present about efficiency in large data models.

Map example

Figure 3-3: Medium memory map example

Item	Usage	Comments
CPU	segmented	–
code	<64K	limited to first segment of 64K.
xnear data	<16K	16K (per task) of fast accessible user data anywhere in 256K/16M via DPP1. This memory space shares DPP1 with the user stack, hence xnear data + user stack < 16K. Use the _xnear keyword.
normal data	>64K	paged data access anywhere in 256K/16M. The size of a single data object is limited to 16K.
near data	<16K	16K (per task) of fast accessible user data anywhere in 256K/16M via 'default data group'. Automatically utilized by **c166** ! The keywords _near, _system and _iram also allow explicit user manipulation.
huge data	allowed	supports huge data access anywhere in 256K/16M. The size of a single huge object is **not** limited to 16K (24 bit address arithmetic). Huge data access is less fast than normal data access. Size of one struct < 64K. Array of struct/any type > 64K
shuge data	allowed	supports shuge data access anywhere in 256K/16M. The size of a single shuge object is limited to 64K (16 bit address arithmetic). Shuge data access is as fast as huge data, but arithmetic on shuge addresses is faster.

Table 3-5: Medium memory model

LANGUAGE

3.2.1.4 LARGE MEMORY MODEL

The compiler does not assume the CSP register to contain something valid. Each call results in a far inter-segment code access (unless the _near keyword is used explicitly in the function prototype). Therefore this model allows code access in all segments up to 16M. As in the medium model, all data accesses are far. The four DPP registers do <u>not</u> contain the system startup values. The DPP registers are used to access the 16M of data in 16K pages. Because the paging principle is used with 14 bit address arithmetic, data objects (e.g. arrays) cannot be greater than 16K (one page), unless the _huge or _shuge keyword is used. The _huge keyword tells the compiler to generate 24 bit address arithmetic. The _shuge keyword tells the compiler to generate 16 bit address arithmetic. Of course the processor must run in segmented mode. In section 3.2.1.7, *Efficiency in Large Data Models (Medium/Large/Huge)* some details are present about efficiency in large data models.
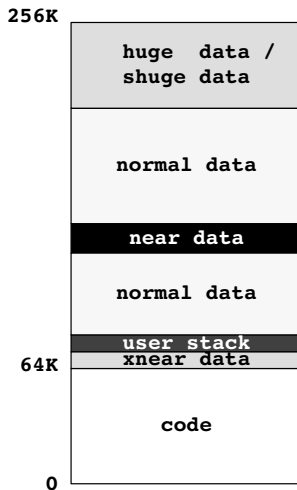
Map example

Figure 3-4: Large memory map example

Item	Usage	Comments
CPU	segmented	–
code	>64K	allows code anywhere in 256K/16M.
normal data	>64K	paged data access anywhere in 256K/16M. The size of a single data object is limited to 16K for objects larger than the specified near data threshold (see the **–T** compiler option).
xnear data	<16K	16K (per task) of fast accessible user data anywhere in 256K/16M via DPP1. This memory space shares DPP1 with the user stack, hence xnear data + user stack < 16K. Use the _xnear keyword.
near data	<16K	16K (per task) of fast accessible user data anywhere in 256K/16M via 'default data group'. Automatically utilized by **c166** ! The keywords _near, _system and _iram also allow explicit user manipulation.
huge data	allowed	supports huge data access anywhere in 256K/16M. The size of a single huge object is **not** limited to 16K (24–bit address arithmetic). Huge data access is less fast than normal data access. Size of one struct < 64K. Array of struct/any type > 64K
shuge data	allowed	supports shuge data access anywhere in 256K/16M. The size of a single shuge object is limited to 64K (16 bit address arithmetic). Shuge data access is as fast as huge data, but arithmetic on shuge addresses is faster.

Table 3-6: Large memory model

LANGUAGE

3.2.1.5 HUGE MEMORY MODEL

The compiler does not assume the CSP register to contain something valid. Each call results in a far inter-segment code access (unless the `_near` keyword is used explicitly in the function prototype). Therefore this model allows code access in all segments up to 16M. All data accesses are huge. The four DPP registers do <u>not</u> contain the system startup values. The DPP registers are used to access the 16M of explicitly far data. Because all accesses are huge by default, data objects can easily be greater than 64K. As with the large and medium models, the processor must run in segmented mode. In section 3.2.1.7, *Efficiency in Large Data Models (Medium/Large/Huge)* some details are present about efficiency in large data models.
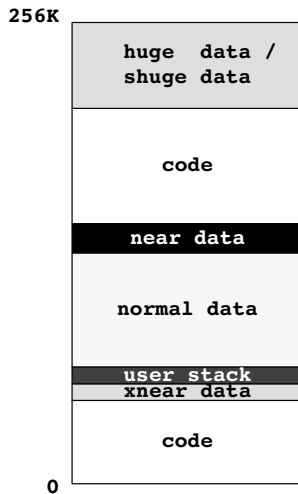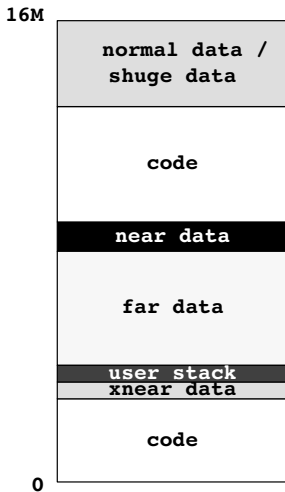
Map example

Figure 3-5: Huge memory map example

Item	Usage	Comments
CPU	segmented	–
code	>64K	allows code anywhere in 256K/16M.
normal data	>64K	huge data access anywhere in 256K/16M. The size of a single object is not limited. Huge data access is less fast than near or far data access. Size of one struct < 64K. Array of struct/any type > 64K
xnear data	<16K	16K (per task) of fast accessible user data anywhere in 256K/16M via DPP1. This memory space shares DPP1 with the user stack, hence xnear data + user stack < 16K. Use the _xnear keyword.
near data	<16K	16K (per task) of fast accessible user data anywhere in 256K/16M via 'default data group'. Automatically utilized by **c166** ! The keywords _near, _system and _iram also allow explicit user manipulation.
far data	>64K	paged data access anywhere in 256K/16M. The size of a single data object is limited to 16K for objects larger than the specified near data threshold (see the **–T** compiler option).
shuge data	allowed	supports shuge data access anywhere in 256K/16M. The size of a single shuge object is limited to 64K (16 bit address arithmetic). Shuge data access is as fast as huge data, but arithmetic on shuge addresses is faster.

Table 3-7: Huge memory model

3.2.1.6 _MODEL

c166 introduces the predefined preprocessor symbol _MODEL. The value of this symbol represents the memory model selected. This can be very helpful in making conditional C code in one source module, used for different applications in different memory models. See also section 3.21, *Portable C Code*, explaining the include file c166.h.

The value of _MODEL is:

tiny model	't'
small model	's'
medium model	'm'
large model	'l'
huge model	'h'

Example:

```
#if _MODEL == 'm' || _MODEL == 'l' /* medium or
                                      large model */
...
#endif
```

3.2.1.7 EFFICIENCY IN LARGE DATA MODELS (MEDIUM/LARGE/HUGE)

For programs compiled with the medium, large and huge memory model, the compiler creates default data sections (member of the default data group) and additional far/huge/shuge data sections for each module. Since accessing data outside the default data page is slower than accessing data within the default data page, programs will run faster if as many of their variables as possible are declared in such a way that they are allocated in the default data page. There are a number of ways to control the allocation of data:

1. All **initialized static/public RAM data** will be allocated in these default data sections unless the _far/_huge/_shuge keyword is explicitly used in the declaration or the **–T** option is used for specifying a certain threshold value for this data.

   All **non–initialized static/public RAM data having a size below a certain 'threshold' value** will be allocated in these default data sections unless the _far/_huge/_shuge keyword is used explicitly in the declaration.

   Strings, floating point constants and jump tables are allocated in ROM and can never be in the default data sections.

The default data sections are member of a special DGROUP group which is (of course) limited to 16K. It is possible to have a DGROUP area (of max 16K) per task. DPP2 is ASSUMED to contain the page number of this group, which is assigned at system startup. During a context switch (interrupt) DPP2, and the scratch register DPP0, are saved, assigned new values and restored afterwards. However, you can also share the default data group area with the default data groups of each task (interrupt).

The sections of the DGROUP must be declared as a COMMON section: same name, same size and same contents. In that case the total size of the default data group area of the whole application is limited to 16K. This results in the following DPP-usage:

DPP0	far pointer dereferencing, external far variables
DPP1	user stack (R0 user stack pointer) / xnear data space
DPP2	default data group (C166_DGROUP)
DPP3	SYSTEM (SFR access, bit-addressable access, iram access and system access)

The threshold value is user definable via the **–T** option. The default value is 256 for non-initialized static/public RAM data. The major advantage of this approach is that better performance is achieved with existing C source code. However, addresses of these variables are still treated 'far', 'huge' or 'shuge' (4 bytes), for usage with (default) pointers.

2. The introduction of the `_near` keyword.
Near forces allocation in the default data group. It also allows better pointer arithmetic, because a pointer to near (2 bytes instead of 4 bytes) is supported. And last but not least near public/external references are supported, assuming DPP2 is used with an external near variable. Of course a near address can be converted to a far, huge or shuge address.

3. The introduction of the `_system` keyword.
System forces allocation in the system data group. The system data group C166_SGROUP is always located in the system page (page 3). It also allows better pointer arithmetic, because a pointer to system (2 bytes instead of 4 bytes) is supported. Public/external references are supported, assuming DPP3 is used with an external system variable. Of course a system address can be converted to a far, huge or shuge address.

LANGUAGE

4. The introduction of the _xnear keyword.
    The **_xnear** keyword forces data to be allocated in the data group
    'C166_XGROUP'. Variables in the 'xnear' memory space have the same
    properties as 'near' variables. The C166_XGROUP contains variables in the
    xnear data space and the user stack. The size of xnear data and the user
    stack size cannot exceed 16Kb. Objects in the xnear data space are
    accessed through DPP1. Of course an xnear address can be converted to a
    far, huge or shuge address.

5. C supports so-called 'tentative declarations', which means that a
    declaration such as 'int i;' remains tentative during the module until
    'defining occurrence' is given (e.g. via 'int i=5;'). If such does not
    happen, it is, for example, allowed to declare this variable to be external
    at the end of the module! Because this programming style is not very
    common (probably only needed for generated C source), the compiler
    option **–Ot** is available, to assign 'defining occurrence' immediately to
    every tentative declaration, allowing more data to be optimized. This
    option is default on, using the medium/large/huge model (lazy
    programmers often 'forget' the static attribute of public non–initialized
    variables which are only used in one module).

    If the tentative property described above is really used in a C program, a
    double definition error will occur. In this case the option must be turned
    off (**–OT**) for this module (or the module must be edited of course).

Using **–OT** results in more code and slower execution.

C166_DGROUP sections

If the cumulated size of all C166_DGROUP sections of a task exceeds 16K,
there are five possibilities to solve it (to be tried in this order):

1. Declare 'near' variables as 'xnear' / 'system' variables.

2. Declare variables to be 'far' explicitly (using the **_far** keyword).

3. Declare variables to be 'huge' explicitly (using the **_huge** keyword).

4. Decrease the 'threshold' values (**–T** option), so more variables are
    allocated in far data sections. If the threshold value is 0, only 'near'
    variables will be allocated in the default data sections.

5. Decrease the number of 'near' variables.

6. Use this possibility only if the other solutions cannot be used!
Use the **–G***groupname* option, to specify the group to be used by the compiler. So, for example, one set of C modules can allocate their default data in the first data group and all other modules allocate their default data in a second data group. If the **–G** option is used, the C compiler emits code at each public (not static) function entry point to preserve the current DPP2 value and assign the page number of the new correct data group to DPP2. At function exit the original DPP2 value is restored. This seems rather expensive, but the gain of code size by using DPP2 can be more than the loss introduced by these instructions.

This is the last alternative and certainly not recommended, because it might introduce some dangerous, hard to find side-effects, as described below in separate notes.

If you use this option, it is your own responsibility to declare 'extern near' variables within the same group! Therefore the compiler emits warnings for 'extern near' declarations if you use the **–G** option.

Be sure that functions called by this module do NOT use their own default data. Some C library functions might use default data too!

3.2.1.8 _NEAR, _XNEAR, _FAR, _HUGE AND _SHUGE

As described before, a limitation of a predefined memory model is that, when you change memory models, all data and code address sizes are subject to change. Therefore **c166** lets you override the default addressing convention for a given memory model and access near, far, huge or shuge objects using special declarations. This is done with the `_near`, `_far`, `_huge` or `_shuge` keyword. These special type modifiers can be used with a standard memory model (except tiny) to overcome addressing limitations for particular items (either data or code) without changing the addressing conventions for the program as a whole.

The `_near`, `_xnear`, `_far`, `_huge` and `_shuge` keywords are not allowed with automatics and parameters (unless used as a target of a pointer of course).

The following explains how the usage of these keywords affects the addressing of code, data or pointers to code or data in all models:

tiny model

In this model all normal data is implicitly `_near`, because the processor does <u>not</u> run in segmented mode. A linear 16 bit (64K) data area is achieved. The `_far`, `_huge` and `_shuge` keywords are not possible (and not allowed).

small model

In this model all normal data is implicitly `_near`. Address arithmetic is performed on 16 bit addresses (linear address space assumed). Therefore objects may be greater than 16K, unless the SND locator control is used, which introduces gaps in the address space of normal data. Besides 64K of normal data (including ROM data), far data is supported. Far data may be anywhere in memory, not assumed to be in the linear data area. You can reference far data using a 24 bit address. Address arithmetic is performed on 14 bit (page offset only). Therefore, individual data items (e.g. arrays) cannot exceed 16K (page) and cannot cross page boundaries if declared `_far`. If you use far objects greater than 16K, you must declare them `_huge` or `_shuge`. Huge data may be anywhere in memory and you can also reference it using a 24 bit address. However, address arithmetic is done using the complete address (24 bit). Shuge data may also be anywhere in memory and you can also reference it using a 24 bit address. However, address arithmetic is done using a 16 bit address.

All function calls are assumed to be `_huge` (maybe in another code segment of 64K). However, an intra-segment call is supported via a `_near` function (the keyword `_near` must be present in the function prototype). In fact you could declare (and define) all static functions as near functions, because they are always allocated in the same code section as the functions they are called by. You cannot apply the `_far` keyword to functions.

medium model

In this model 'near data' means data allocated into a special page for fast access. See section 3.2.1.7, *Efficiency in Large Data Models (Medium/Large/Huge)* for more details on the 'default data group'. Address arithmetic on near and far data is always 14 bit. As in the small model, huge and shuge data access is supported.

This model also supports 'xnear' data. This data is allocated together with the user stack in DPP1. The access to this memory space is just as fast as to 'near' data. Address arithmetic on `_xnear` data is done in 14 bits. See section 3.2.1.7, *Efficiency in Large Data Models (Medium/Large/Huge)* for more details on the 'C166_XGROUP' data group.

• • • • • • • • •

All function calls are assumed to be in the same (first) segment of 64K. However, an inter-segment call is supported via a huge function (the keyword _huge must be present in the function prototype). The _huge function may not call any standard C library function, run-time library or any normal _near function in another segment. You cannot apply the _far keyword to functions.

large/huge model

In these models 'near data' means data allocated into a special page for fast access. See section 3.2.1.7, *Efficiency in Large Data Models (Medium/Large/Huge)* for more details on the 'default data group'. Address arithmetic on near and far data is always 14 bit. As in the small and medium models, huge and shuge data access is supported.

Without any of the _near, _xnear, _far, _huge and _shuge keywords, the default data access is _far paged data for the large model and _huge for the huge model.

These models also support 'xnear' data. This data is allocated together with the user stack in DPP1. The access to this memory space is just as fast as to 'near' data. Address arithmetic on _xnear data is done in 14 bits. See section 3.2.1.7, *Efficiency in Large Data Models (Medium/Large/Huge)* for more details on the 'C166_XGROUP' data group.

All function calls are assumed to be _huge (in another code segment of 64K), unless you use the _near keyword in the function prototype. In fact you could declare (and define) all static functions as near functions, because they are always allocated in the same code section as the functions they are called by.

Object or pointer modification

The _near, _xnear, _far, _huge and _shuge keywords modify either objects or pointers to objects. When using them to declare data or code (or pointers to data or code), the following rules must be kept in mind:

- The keyword always modifies the object or pointer immediately to its right. In complex declarations such as

      char _far * _near p;

  think of the _far keyword and the item to its right as being a single unit. In this case, p is a pointer to a far char, and therefore contains a 24 bit far address.

- If the item immediately to the right of the keyword is an identifier, the keyword determines the storage type of the item: whether it must be allocated in the default data section or a separate data section. In this case the pointer **p** is explicitly declared to be allocated in normal data (if tiny/small model is used) or in the default data group (if medium/large/huge model is used).

- If the item immediately to the right of the keyword is a pointer (a '*' (star)), the keyword determines the logical type: whether the pointer will hold a **_near** address (2 bytes), a **_far** address (4 bytes), a **_huge** address (4 bytes) or an **_shuge** address (4 bytes). For example,

        char   _far   * _near p;

    allocates **p** as a **_far** pointer to an item of type char. The pointer **p** itself is allocated in near data.

- The memory model used determines the default logical type of a pointer. In:

        int *p;

    p is a far pointer when you use the medium or large model, a huge pointer in the huge model otherwise a near pointer. The storage type of **p** itself is near in tiny and small model, and, depending on the threshold value, probably also near in medium, large and huge model.

- You cannot apply the **_far** keyword to functions.

3.2.1.9 _SYSTEM, _IRAM AND _BITA

As described before, **c166** lets you override the default addressing convention for a given memory model and access near, far, huge or shuge objects using special declarations. But also special declarations are supported by **c166** to access data objects in the SYSTEM page, like internal RAM data, overall system data or bitaddressable memory. This is done with the keywords **_system**, **_iram** and **_bita**. These special type modifiers can be used in all memory models to overcome addressing 'limitations' for particular near data items.

The **_system**, **_iram** and **_bita** keywords are not allowed with automatics, functions and constants unless used as a target of a pointer.

• • • • • • • • •

_system

Objects declared with the keyword **_system** are allocated in system data sections (see paragraph 3.2.3, *Section Allocation*). The system data sections are member of the special group C166_SGROUP which is limited to the size of the SYSTEM page (16K-SFRs). DPP3 is ASSUMED to contain the page number of this group which is equal to the SYSTEM page number (page 3) and is assigned at system startup.

_iram

Objects declared with the keyword **_iram** are allocated in IRAMADDRESSABLE data sections (see paragraph 3.2.3, *Section Allocation*). The locator places IRAMADDRESSABLE sections in the internal RAM of the C166/ST10.

Addressing of _iram objects is exactly the same as addressing **_system** objects because the internal RAM is located in the SYSTEM page. Both _iram and **_system** are addressed via the SYSTEM data page pointer DPP3 which is assigned to the system page at system startup.

The _iram sections are limited to 2048 bytes internal RAM. By default the _iram section size is limited by the compiler to 2048 bytes. But you can always set your own _iram sections size limit with the **–m** *mem=size* compiler option (e.g. **–mIR=512**). See for more information section 4.3, *Detailed Description of the Compiler Options*.

_bita

When using bit fields in structures that are located in bitaddressable memory the compiler can take advantage of the bit and bit field instructions of the processor. You can tell the compiler that a struct must be located in bitaddressable memory by using the **_bita** memory qualifier.

Example:

```
_bita struct {
     unsigned  bf1:1;
     unsigned  pit:2;
     unsigned  bf2:1;
} s;
```

The compiler will allocate the struct in a bitaddressable section. For nested structures and unions _bita can only be applied to the outer level. When _bita is used for structure members the compiler ignores this.

Example:

```
struct m {
     int  m1:2;
     int  m2:3;
} mm;

struct n {
     _bita struct m  n1;  // _bita ignored
     struct m  n2;
} nn;
```

Even with the _bita keyword structures will be word aligned. Also the structure members are aligned as they would be without the _bita qualifier; i.e., byte addressable members (signed/unsigned char) are byte aligned and word addressable members (such as int and pointers) are word aligned.

The _bita keyword can also be applied to global or static variables of type char, int and long. In bitaddressable memory chars will be word aligned. When accessing single bits in these variables like:

```
_bita int w;

w |= 0x4000;
if (w & (1 << 10))
{
     w &= 0xFFEF;
}
```

then the compiler will use bit instructions:

```
     BSET _w.14
     JNB  _w.10,_3
     BCLR _w.4
_3:
```

For non-static local variables the _bita keyword is not allowed. Most local variables will be placed in registers automatically, making them bitaddressable anyway. See also the pragmas **autobita** and **autobitastruct** in section 4.5, *Pragmas*.

3.2.2   USER STACK MODEL

If you use the **–P** or **–Pd** option of **c166**, the compiler does not emit the regular CALL/RET instructions when calling a C function, but emits code using a jumping mechanism, specifying the return address on the user stack. The advantage of this approach is that the system stack is not used at all. The price paid for this feature is an execution speed penalty.

In EDE you can select the user stack model as follows:
From the **Project** menu, select **Project Options...** Expand the **Application** entry and select **Memory Model**. Enable the **Use user stack for return addresses** check box.

When using plain user stack model, special libraries are needed to support this feature. These user stack model libraries are an integral part of this product. If **–Pd** was specified at the command line, all calls to the library use the regular CALL/RET calling convention.

This behavior can also be forced for user defined functions using either the _usm or _nousm function qualifiers. If _usm is specified at the function definition, the function is called using user stack model calling conventions. If _nousm is specified, the function is called using the generic CALL/RET calling method, even if **–P** was specified on the command line.

–P option	Libraries	Def. func. qualifier	_USMLIB macro
none	default	_nousm	_nousm
–P	USM	_usm	_usm
–Pd	default	_usm	_nousm

Table 3-8: User stack model

There are two valid reasons to use this option (and libraries):

• Real-time Operation Systems

  When using a real-time kernel, it is often not allowed to use the system stack area (in fact change SP), because this area is reserved for the kernel. Therefore, the **–P** option can be used, when using a kernel. Please refer to the documentation supplied with the kernel to verify if this option must be used.

LANGUAGE

- Heavy recursion

  When the system stack area is getting too small and it is not possible to implement a circular system stack approach (using SOV/SUN exception handlers), the **–P** option can be used. In this case the compiler uses the user stack instead of the system stack. You must link the application with the user stack model libraries.

Using **–P** does not mean that you have to use a kernel. You can run the application as a standalone application, without any kernel.

For more details see Appendix D, *User Stack Model*.

3.2.3   SECTION ALLOCATION

Unlike some other microcontrollers, the C166/ST10 microcontroller does not have different memory spaces with the same address. This means that a non–automatic object can be referred to solely by its starting address, because the address represents a unique memory location. There is also no difference in assembly code accessing internal RAM, external RAM, internal ROM or external ROM (within the same page/segment).

The processor, however, distinguishes memory access in execution speed. Code access to internal ROM is faster than access to external ROM. Data access to internal RAM is faster than access to external RAM. So, a piece of assembly code executes faster if the code is allocated in internal ROM instead of external ROM. And the same piece of code gets an even higher execution speed if the data structures accessed are allocated in internal RAM instead of external RAM.

In the C166/ST10 compiler the code generator does not have to know if internal or external RAM is accessed, because the same code can be generated. Execution speed is in fact a matter of allocating sections in internal memory instead of external memory. The allocation of sections is done by the locator stage of **l166**, and can be manipulated by specifying a memory range for each 'class' of sections.

c166 allows you to control the class, align type and combine type of a section with a command line option (e.g. **–RclNB=NEARRAM** changes the class of non–initialized near data to 'NEARRAM' for this module). The disadvantage of this method is that the changed attributes are used for the complete C module.

However, using pragmas, **c166** allows more flexibility of storage specification within a C module. In this approach it is possible to declare for example only a few C variables of a module to be allocated in a special section which must be PEC–addressable and the rest in normal data sections. Or only one function of the module in internal ROM and the rest in external ROM.

LANGUAGE

Naming convention

c166 uses a naming convention for the generated sections. In general the following modifications are applied to a filename:

- whitespace and dots are converted to underscores
- filenames are converted to uppercase.
- if a filename starts with a digit, the first digit is replaced by an underscore.

Everything after (and including) the last dot is stripped from the filename. Thus, the filename: "`long file.name.c`" will result in the following string to be used as a basis for the section name (in the text below referred to as "*module*"):

"`LONG_FILE_NAME`"

The length of a filename is unlimited. Furthermore, the section naming is divided into three categories as described below:

I Non-initialized Data Sections/Normal Sections/Romdata Sections

For non-initialized data sections, normal sections and romdata sections the section name is generated as follows:

module_number_mem

where,

module   is the module name in uppercase (without suffix) of the `.c` file

number   is a unique number.

mem   is a memory abbreviation code as shown in the next table.

You may change the section attributes of this category.

c166 uses the following table for its defaults (e.g. compiling `mod.c`):

Description	mem	type		align	combine	class	example
		–Mm/ –Ml	**–Mt/ –Ms**	type	type		section name
bits	BI	BIT	BIT	BIT	PUBLIC	CBITS	MOD_1_BI
strings/floating point constants[1]	CO	DATA	LDAT	WORD	PUBLIC	CROM	MOD_2_CO
bitwords	BA	DATA	LDAT	WORD	PUBLIC	CBITWORDS	MOD_3_BA

• • • • • • • • •

Description	mem	type		align	combine	class	example
		–Mm/ –Ml	**–Mt/ –Ms**	**type**	**type**		**section name**
near data	NB	DATA	LDAT	WORD	PUBLIC	CNEAR	MOD_4_NB
xnear data	XN	DATA	––	WORD	PUBLIC	CUSTACK	MOD_15_XN
far data	FB	DATA	PDAT	WORD	PUBLIC	CFAR	MOD_5_FB
huge data	HB	HDAT	HDAT	WORD	PUBLIC	CHUGE	MOD_6_HB
shuge data	XB	SDAT	SDAT	WORD	PUBLIC	CSHUGE	MOD_7_XB
functions	PR	CODE	CODE	WORD	PUBLIC	CPROGRAM	MOD_8_PR
near romdata	NC	DATA	LDAT	WORD	PUBLIC	CNEAR[2]	MOD_9_NC
xnear romdata	XR	DATA	––	WORD	PUBLIC	CUSTACK	MOD_16_XR
far romdata	FC	DATA	PDAT	WORD	PUBLIC	CFARROM	MOD_10_FC
huge romdata	HC	HDAT	HDAT	WORD	PUBLIC	CHUGEROM	MOD_11_HC
shuge romdata	XC	SDAT	SDAT	WORD	PUBLIC	CSHUGEROM	MOD_12_XC
system data	SB	DATA	DATA	WORD	PUBLIC	CSYSTEM	MOD_12_SB
internal ram data	IR	DATA	LDAT	IRAM– ADDRES– SABLE	PUBLIC	CIRAM	MOD_14_IR

[1] See also section 3.2.5, *Constant Romdata Section Allocation*, for small model only.

[2] CNEARROM when tiny/small model is used.

Table 3-9: Section names (non-initialized data, normal and romdata)

When using the medium or large model, near data, xnear data or system data always remain a member of the default data group or system data group. So for these memory areas, it is not possible to change all section attributes.

II *Initialized Ramdata Sections*

For initialized data the section name is generated as follows:

>   *module*_IR_*mem*
>   *module*_ID_*mem*
>   *module*_ER_*mem*
>   *module*_ED_*mem*

where,

module   is the module name in uppercase (without suffix) of the **.c** file

mem   is a memory abbreviation code as used by non–initialized ramdata sections (SB, IR, BI, BA, NB, FB, HB or XB).

You can NOT change the section attributes of this category.

c166 uses the following table for its defaults (near data):

Description	type		align–	combine	class	example
	–Mm/ –Ml	**–Mt/ –Ms**	**type**	**type**		**section name**
near iramdata (ROM copy)	DATA	LDAT (t) PDAT (s)	WORD	PUBLIC	CINITROM	MOD_IR_NB
near iramdata (RAM space)	DATA	LDAT	WORD	PUBLIC	CINITIRAM	MOD_ID_NB
near eramdata (ROM copy)	DATA	LDAT (t) PDAT (s)	WORD	PUBLIC	CINITROM	MOD_ER_NB
near eramdata (RAM space)	DATA	LDAT	WORD	PUBLIC	CINITERAM	MOD_ED_NB

Table 3-10: Section names (initialized romdata)

Example:

File **mod.c** contains the following initialized romdata:

```
#pragma eramdata
int i = 1;      /* default near data */

#pragma iramdata
_far int j = 2;
```

Generated assembly (compiled with **–Ms**):

```
MOD_ER_NB       SECTION PDAT WORD PUBLIC 'CINITROM'
MOD_ER_NB_ENTRY LABEL   BYTE
        DW      01h
MOD_ER_NB       ENDS

MOD_ED_NB       SECTION LDAT WORD PUBLIC 'CINITERAM'
MOD_ED_NB_ENTRY LABEL   BYTE
_i      LABEL   WORD
        DS      2
        PUBLIC  _i
MOD_ED_NB       ENDS
```

• • • • • • • • •

```
          MOD_IR_FB        SECTION PDAT WORD PUBLIC 'CINITROM'
          MOD_IR_FB_ENTRY LABEL    BYTE
                  DW       02h
          MOD_IR_FB        ENDS

          MOD_ID_FB        SECTION PDAT WORD PUBLIC 'CINITIRAM'
          MOD_ID_FB_ENTRY LABEL    BYTE
          _j      LABEL    WORD
                  DS       2
                  PUBLIC   _j
          MOD_ID_FB        ENDS
```

III Specials

The following special section names exist:

C166_INIT	init table for initialized RAM
C166_BSS	clear table for non-initialized RAM
C166_US	user stack
C166_US0	user stack for local register bank 0.
C166_US1	user stack for local register bank 1.
C166_INT	scalable interrupt vector table.
?C166_HEAP	heap section for memory allocation
	(linker or locator generated)
?INTVECT	interrupt vector table (locator generated)

You can NOT change the section attributes of this category.

c166 uses the following table for its defaults:

Description	type		align–	combine	class	(fixed)
	–Mm / –Ml	–Mt / –Ms	type	type		section name
user stack	DATA	LDAT	WORD	GLBUSRSTACK	CUSTACK	C166_US
user stack	DATA	LDAT	WORD	GLBUSRSTACK	CUSTACK	C166_US0
user stack	DATA	LDAT	WORD	GLBUSRSTACK	CUSTACK	C166_US1
init table	DATA	LDAT (t) PDAT (s)	WORD	GLOBAL	CINITROM	C166_INIT
clear table	DATA	LDAT (t) PDAT (s)	WORD	GLOBAL	CINITROM	C166_BSS
heap	HDAT	LDAT	WORD	PUBLIC	?CHEAP	?C166_HEAP
vector table	CODE	CODE	WORD	PUBLIC	C166_VECTAB	C166_INT

Table 3-11: Section names (specials)

You can only change the section attributes of non-initialized data sections, normal sections and romdata sections (category I), using the *mem* code listed in the table.

You can tell the compiler to use other class names, combine types and align types instead of the defaults listed above by means of the following pragmas. Each pragma, has an equivalent command line option that can be used if the complete module must use the changed attributes.

```
#pragma class mem=name       /*    use name as class for
                                    section of area mem */
#pragma combine mem=ctype    /*    use ctype as combine type
                                    for section of area mem */
#pragma align mem=atype      /*    use atype as align type
                                    for section of area mem */
#pragma default_attributes   /*    use default attributes as
                                    listed above */
```

atype is one of the following align types:

B	Byte alignment
W	Word alignment
D	Double word alignment
P	Page alignment
S	Segment alignment
C	PEC addressable
I	IRAM addressable

ctype is one of the following combine types:

L	private ('Local')
P	Public
C	Common
G	Global
S	Sysstack
U	Usrstack
A *address*	Absolute section AT constant *address* (decimal, octal or hexadecimal number)

Examples:

1. The C module is called 'test.c'. The example illustrates how to allocate one array in a special section with the class 'SLOWRAM' and the rest of the data in data section with default attributes. The generated code is listed below:

   C:

   ```
   #pragma class nb=SLOWRAM
   int array[1000];

   #pragma default_attributes
   int j;
   ```

   Generated code:

   ```
   TEST_1_NB  SECTION  LDAT WORD PUBLIC 'SLOWRAM'
   TEST_1_NB_ENTRY  LABEL  BYTE
   _array  LABEL  WORD
         DS     2000
         PUBLIC _array
   TEST_1_NB  ENDS

   TEST_2_NB  SECTION  LDAT WORD PUBLIC 'CNEAR'
   TEST_2_NB_ENTRY  LABEL  BYTE
   _j    LABEL WORD
         DS     2
         PUBLIC _j
   TEST_2_NB  ENDS
   ```

2. The C module is called 'test.c'. The example illustrates how to allocate one C variable on a fixed memory location (address 8000H) and the rest of the data in a data section with default attributes. As described in the 'TASKING C166/ST10 Cross-Assembler, Linker/Locator, Utilities User's Manual', AT is considered as an additional align-type and implies the default combine type PRIVATE.

   C:

   ```
   #pragma combine nb=A32768
   volatile int      cntrl_reg;
         /* e.g. an I/O register of peripheral chip */

   #pragma default_attributes
   int i;
   ```

LANGUAGE

Generated code:

```
TEST_1_NB  SECTION  LDAT WORD AT 08000h 'CNEAR'
TEST_1_NB_ENTRY  LABEL  BYTE
_cntrl_reg  LABEL  WORD
      DS    2
      PUBLIC _cntrl_reg
TEST_1_NB  ENDS

TEST_2_NB  SECTION  LDAT WORD PUBLIC 'CNEAR'
TEST_2_NB_ENTRY   LABEL BYTE
_i    LABEL WORD
      DS    2
      PUBLIC _i
TEST_2_NB  ENDS
```

3.2.4   CODE MEMORY FRAGMENTATION

By default the compiler uses one section per module that contains the code. You can change this behavior with the following pragmas:

```
#pragma fragment
#pragma fragment resume
#pragma fragment continue
```

The **#pragma fragment** causes the compiler to generate each single function in its own section. The compiler will continue to do so until it encounters either **#pragma fragment resume** or **#pragma fragment continue**.

In case of **#pragma fragment resume** the compiler will resume code generation in the last active section (with the same attributes) before **#pragma fragment**.

In case of **#pragma fragment continue** the compiler will start a new continuous code.

These pragmas are especially useful in combination with the smart linking feature of the linker/locator. When you use smart linking, the linker will only link sections that are referenced. Thus if each function has its own section, only functions that are actually called (referenced) are linked rather than all functions in an `.obj` file at once.

Example:

```
void func1( void ) { }          /* Code section 1           */

#pragma fragment
void func2( void ) { }          /* Code section 2           */
void func3( void ) { }          /* Code section 3           */

#pragma fragment resume
void func4( void ) { }          /* Resume in code section 1  */

#pragma fragment
void func5( void ) { }          /* Code section 4           */

#pragma fragment continue
void func6( void ) { }          /* Continue in code section 5 */
void _near func7( void ) { }  /* Code section 5           */

#pragma fragment resume         /* No effect: Code section 5  */
void func8( void ) { }
#pragma fragment continue       /* No effect                */

#pragma fragment
_near void func9( void ) { }  /* Code section 6           */

#pragma fragment resume
void main( void )               /* Resume in code section 5  */
{
    func9();
    func7();
    return;
}
```

3.2.5   CONSTANT ROMDATA SECTION ALLOCATION

In the small memory model **c166** default allocates all constant romdata for strings, floating point constants, initialization of aggregates and jump tables in normal data (near in small memory model), which is limited to 4 pages of 16K. When you do not want to sacrifice a normal data page for ROM, you should use the **–Oe** option of **c166**.

When the **–Oe** option is enabled the following changes are in effect for the small memory model:

- **c166** allocates string and floating point constants in a far romdata section (PDAT). During startup this data is copied from far ROM to near RAM like initialized ramdata. The code generated for accessing these constants is not changed. This means no change in execution speed. The disadvantage is that the memory for these constants is allocated twice: once in far ROM and once in near RAM. The ROM sections have class 'CINITROM' and the RAM sections have the class 'CINITERAM' or 'CINITIRAM', depending on the **#pragma eramdata/iramdata**.

- constant data for initialization of automatic aggregates and jump tables is allocated in far ROM. **c166** generates different code for accessing this data as far data, which implies a minor draw-back in code execution performance.

When you use the **const** keyword for normal data, this data is placed in near ROM, even with the **–Oe** option.

To move jump tables separately from string and floating point constants to various locations, you can use the following pragmas:

#pragma switch_tabmem_far

For the small memory model, jump tables are placed in far ROM. The location of string and floating point constants is still controlled by the **–Oe**/**–OE** option as described above. The ROM section where the jump tables are placed have class 'CFARROM'. The code generated for accessing the jump table in far ROM is slightly slower compared to the situation where jump tables reside in near ROM.

#pragma switch_tabmem_near

For the small memory model, jump tables are placed in near ROM. The location of string and floating point constants is still controlled by the **–Oe** / **–OE** option as described above. The ROM section where the jump tables are placed have class 'CNEARROM'.

#pragma switch_tabmem_default

This is the default. Use this pragma to return the control of the jump table locations back to the **–Oe** / **–OE** command line option as described above.

The pragmas **switch_tabmem_far**, **switch_tabmem_near** and **switch_tabmem_default** can be used anywhere in the source file. The location of the jump table is affected by the last pragma before a switch statement.

The pragmas can be passed through the command line by using the **–z**_pragma_ command line option.

The delivered small C libraries do not support constant romdata as far data, because it is not commonly used. All C library functions are compiled with the default option **–OE**, to allocate constant romdata 'CROM' in linear data sections (LDAT). You have to re-compile the C-library functions which contain constant romdata 'CROM' with the option **–Oe** if you do not want near ROM. You can rebuild the small C libraries (`c166s.lib` and `c166ss.lib`) using the makefiles in the library directories.

All library modules are re-compiled and the libraries are rebuilt by these makefiles.

String constants are in:

```
_doflt.c,   _dowflt.c, _doprint.c, _dowprin.c,
_doscan.c,  _dowscan.c, _tzone.c,   asctime.c,
assert.h,   fss_init.c, locale.c,   perror.c,
raise.c,    strerror.c, strftime.c, tmpfile.c,
tmpnam.c,   wcsftime.c, wctrans.c,  wctype.c
```

The **const** keyword is in:

```
_ctype.c, strftime.c, wcsftime.c
```

Floating point constants are in:

```
_atan.c,    _doflt.c, _dowflt.c, _fmod.c,   _getflt.c,
_getwflt.c, _sinus.c, _strtod.c, _wcstod.c, acos.c,
asin.c,     atan.c,   atan2.c,   cos.c,     cosh.c,
exp.c,      log.c,    log10.c,   pow.c,     sinh.c,
sqrt.c,     strtod.c, tan.c,     tanh.c,    wcstod.c
```

Before running these makefile you should have rights to write to the library files `c166s.lib` and `c166ss.lib`.

Restriction:

When the **#pragma initeram** or **#pragma initiram** is used, only the last pragma in the source file affects the section attributes of the near ram data sections for string and floating point constants.

LANGUAGE

3.2.6   THE _AT( ) ATTRIBUTE

In **c166** it is possible to locate a global variable at a specified address. This can be done with the _at() attribute. The syntax is:

    _at(*address*)

where, *address* is the location in memory of the variable.

In the tiny memory model, the address is limited to 64Kbytes. In all other models, the address space of the used device is the limit.

The _at() attribute can only be used on non-initialized global variables. Variables, which are declared constant, using the const modifier can be initialized and they will be placed in a rom section. Depending on the memory modifier, this will be near-, far-, huge- or shugerom.

If a variable meets the **autobita** or **autobitastruct** pragma requirements and the _at() keyword is specified, the _at() attribute overrules the **autobita**/**autobitastruct** pragmas.

The _at() attribute has no effect on variables which are declared extern.

In the segmented memory models, variables which have the _at() attribute are not moved automatically to near memory. However, you can explicitly specify an absolute variable to be near.

For near variables, the locator automatically assigns the correct page to the correct DPP register. Note that all other relocatable variables in the concerning page will also be moved. The dynamic assignments of DPP registers can be overruled by the linker/locator controls. However, in case of absolute variables, this will usually lead to errors because there is only one valid DPP-register / page-number combination.

If two sections overlap, or if not all near sections can be located the linker/locator will generate an error message.

The _at() attribute cannot be used with the _bit, _system, _bita, _sfr, _esfr, _xsfr and _iram memory modifiers.

Examples:

```
_near int i _at(0x29000);
_far const char ch _at(0x2A900) = 100;
int j, * k _at(0x2B002);
int * (* * fptr)(int, int) _at(0x12344);
```

This will generate the following sections, when compiled in the small memory model:

```
TEST_1_NB  SECTION  LDAT WORD AT 029000h 'CNEAR'
TEST_1_NB_ENTRY     LABEL  BYTE
_i    LABEL WORD
      DS    2
      PUBLIC _i
TEST_1_NB  ENDS

TEST_2_FC  SECTION  PDAT BYTE AT 02A900h 'CFARROM'
TEST_2_FC_ENTRY
_ch   LABEL BYTE
      DB    64h
      PUBLIC _ch
TEST_2_FC  ENDS

TEST_3_NB  SECTION LDAT WORD AT 02B002h 'CNEAR'
TEST_3_NB_ENTRY  LABEL  BYTE
_k    LABEL WORD
      DS    2
      PUBLIC _k
TEST_3_NB  ENDS

TEST_4_NB  SECTION LDAT WORD AT 012344h 'CNEAR'
TEST_4_NB_ENTRY  LABEL  BYTE
_fptr LABEL WORD
      DS    2
      PUBLIC _fptr
TEST_4__NB  ENDS

TEST_5_NB  SECTION LDAT WORD PUBLIC 'CNEAR'
TEST_5_NB_ENTRY  LABEL  BYTE
_j    LABEL WORD
      DS    2
      PUBLIC _j
TEST_5_NB  ENDS
```

For example, in this case the linker/locator assigns a value of 0x0A to DPP2. This is the same as using the SND(DPP2(10)) linker/locator control.

When specifying a near address, bits 14 and 15 implicitly specify the DPP-register that will be used. DPP3 cannot be changed. This is because DPP3 points to the memory that contains SFRs and bit addressable memory.Therefore it is not possible to locate 'near' variables in the third page of any segment, other than segment 0.

3.2.7   THE _ATBIT( ) ATTRIBUTE

In **c166** it is possible to define bit variables within a `_bitword` or
(bit-addressable) _sfr variable. This can be done with the `_atbit()`
attribute. The syntax is:

```
_atbit( name, offset )
```

where, *name* is the name of a `_bitword` or `_sfr` variable and *offset*
(range 0-15) is the bit-offset within the variable.

Examples:

```
_sfr  P0;
_sfrbit   P0_6  _atbit( P0, 6 );

_bitword  bw;  /* bitaddressable word */
_bit      myb   _atbit( bw, 3 );
```

Using the defined bit:

```
if ( myb )
    myb = 0;
```

generates the same code as:

```
if ( _getbit( bw, 3 ) )
    _putbit( 0, bw, 3 );
```

The first example defines an `_sfrbit` within a (bit-addressable) `_sfr`
variable. The second example defines a bitaddress within a bitaddressable
word. For more information on SFR variables see section 3.4.5, *Special
Function Registers*. For more information on `_bitword` variables see
section 3.4.4, *The Bitword Type*.

The storage class of the defined bit is ignored. The storage class is
inherited from the `_bitword` variable instead.

The `_atbit()` attribute has no effect on variables which are declared
extern.

In the following situation, the bit `b0` will be allocated statically. Yet it will be initialized at run-time, each time the function is entered:

```
_bit funct ( void )
{
    static _bitword bw;
    _bit b0 _atbit( bw, 2 ) = 1;
    return b0;
}
```

3.2.8   INLINE C FUNCTIONS: _inline

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.

Example:

```
_inline int
add( int a, int b )
{
    return( a + b );
}

void
main( void )
{
    int c = add( 1, 2 );
}
```

The pragmas `asm` and `endasm` are allowed in inline functions. This makes it possible to define inline assembly functions. See also section 3.11, *Inline Assembly* in this chapter.

LANGUAGE

3.2.9   UNALIGNED DATA: _noalign

With the `_noalign` attribute you can tell the compiler that an object is
possibly located at an unaligned address and that the compiler must not
spend any effort to align the data. This means that the object will not be
aligned using an EVEN directive. When the `_noalign` attribute is applied
to a struct/union member, the member will not be aligned. When an
object at a possibly unaligned address needs to be accessed, the compiler
generates two byte instructions. The following example illustrates this.

```
_noalign int i;      /* possibly unaligned object */

void func( void )
{
    i++;
    return;
}
```

This generates the following code:

```
_func    PROC    FAR
         MOVB    RL1,_i          ;; fetch object using
         MOVB    RH1,(_i+1)      ;; byte instructions
         ADD     R1,#01h
         MOVB    _i,RL1          ;; store object using
         MOVB    (_i+1),RH1      ;; byte instructions
         RETS
_func    ENDP
```

Since the code that is generated is less efficient (two byte instructions), use
this attribute only when really needed. For example, for data exchange
with 8-bit processors.

Do not use the `_noalign` attribute on:

- automatic/register variables
- parameters
- function return values

The compiler issues a warning when you use the `_noalign` attribute on
an unsupported object.

It is also possible to have a pointer referring to an unaligned object. In this case the type the pointer refers to must also be qualified as possibly unaligned. For example,

```
_noalign int   i;           /* possibly unaligned int */
_noalign int * pi;          /* pointer referring to
                               possibly unaligned int */
_noalign int * _noalign npi;  /* same as above, but
                                 pointer itself is
                                 also unaligned    */
```

3.2.10 USING PACKED STRUCTURES: _packed

By default the compiler aligns structure members on word boundaries. Due to this alignment 'gaps' can appear between the structure members. When you do not want these gaps, you can use the _packed qualifier. In this case the compiler does not align the structure members. However, bit fields will still be aligned in some special cases, as explained below. With the _packed attribute the compiler pads the size of the structure or union to an 8-bit boundary instead of a 16-bit boundary.

You can use the _packed attribute on struct and union types only. The _packed attribute applies to the struct/union definition itself, rather than to an instance of the struct/union. So, each instance of the struct/union must also have the _packed attribute. The following example demonstrates the usage of the _packed attribute:

```
_packed struct ps
{
    char    c;              /* offset 0 bytes */
    int     i;              /* offset 1 byte  */
};

_packed struct ps st0;  /* correct, _packed struct */
        struct ps st1;  /* error, conflict in
                            _packed attribute       */
```

LANGUAGE

Besides the padding of the struct/union size, the **_packed** attribute is basically a shortcut for:

```
struct s
{
    _noalign char   c;
    _noalign int    i;
} st2;
```

So, if you want to use a pointer to a member of a **_packed** struct/union, you must qualify the pointer with the **_noalign** attribute:

```
 _noalign int * p = &st2.i;
```

The **_packed** attribute does not say anything about the alignment of the struct/union itself. Therefore, when an instance of a struct/union does not need alignment you must add the **_noalign** attribute:

```
typedef _packed struct ps
{
    char    c;      /* offset 0 bytes       */
    int     i;      /* offset 1 byte        */
    char    byte;   /* offset 3 bytes       */
} tPS;              /* struct size: 4 bytes */


        tPS st3;  /* aligned _packed structure     */
_noalign tPS st4;  /* not aligned _packed structure */
```

When you do not use the **_noalign** attribute on **_packed** structures, the compiler can use the word copy routines for **_packed** struct/unions.

Since the code that is generated after the **_packed** qualifier is less efficient, use packed structures only when really needed, for example for data exchange with 8-bit processors. Consider in such case first other solutions like for example, mapping structures on character arrays.

Bit fields in packed structures

Bit fields in a `_packed` struct/union sometimes need alignment. The following example shows such a situation:

```
_packed struct ps
{
    int  bf7  :  7;  /* bit-offset 00-06      */
                     /* gap: 1 bit            */
    int  bf10 : 10;  /* bit-offset 08-17      */
                     /* bit-offset 18-23:
                        padding to next byte  */
};
```

In this example, there is a gap of one bit between `bf7` and `bf10` and the total size of the structure is 3 bytes. The alignment is needed because otherwise 3 byte moves are needed in order to access all bits of `bf10`.

In the next situation no alignment is needed:

```
_packed struct  ps
{
    int  bf9  : 9;   /* bit-offset 00-08      */
    int  bf7  : 7;   /* bit-offset 09-15      */
                     /* no padding needed     */
};
```

The latter example does not need alignment nor padding. The size of the structure is 2 bytes.

Note that an unnamed bit field with size 0 aligns to the next word boundary as is the case with non-packed struct/unions.

3.3   TASK SCOPE

c166 supports both the 'Task Concept' and the 'Flat Interrupt Concept'. These two concepts are explained in the chapter *Software Concept* of the 'TASKING Cross-Assembler, Linker/Locator, Utilities User's Manual'. We strongly recommend reading this section first!

When the Task Concept is strictly followed, the entry point of each task is an interrupt function, either activated by hardware (interrupt) or by software (TRAP instruction). Each task has only one entry point and no code and data is shared. This implies that reentrancy of code does not exist. See section 3.12, *Interrupt* in this chapter for more details about interrupt functions.

In C the outermost level of scope is a public (non-static) variable. Via the **extern** keyword this variable can be accessed in other C modules. This scope level in C is treated by **c166** as the task scope (public) in the Task Concept. This means that all public/extern variables are not known outside the task. This allows each task to have its own I/O channels and administration (e.g. `printf()`), heap area (e.g. `malloc()`), floating point stack and public data. The public/extern variables are solved at the link stage of **l166**. In practice it is in a lot of cases possible to share code and data between several tasks or interrupt functions. The following ways exist to do this:

define code or data to be shared to 'COMMON'

In this case, the common section must be linked with each task needing access to the shared data/code. The 'COMMON' section attribute tells the locator to 'overlay' the section with another common section carrying the same name. The module referencing the shared data of another C module uses the normal keyword **extern** in the declaration. When using, a prototype of the function is enough. Similar to the normal C rules, the **extern** keyword may be omitted with functions. This approach is used by the C library, where a number of standard C functions (such as `strlen()` and `isdigit()`) are allocated in common sections. The ROM table used by `<ctype.h>` functions is allocated in a common data section. Therefore, the C library must be linked with each task.

The combine type of a section can be changed in two ways. Firstly a command line option (**–R**), resulting in shared code and data of the complete C module. Secondly via a pragma, allowing some data or code of a C module to be shared and the rest not.

• • • • • • • • •

Example:

C module is called `test.c`. The example illustrates how to declare a ROM table (array) as 'shared among several tasks' and the rest of the C data in a normal data section. The generated code is listed below.

```
#pragma save_attributes
#if _MODEL == 'l' || _MODEL == 'm'
#pragma combine fc=C
#define FAR _far /* far common data */
#else
#pragma combine nc=C
#define FAR      /* normal common data */
#endif

/*
 *   COMMON data section in ROM, linked with
 *   each task and overlaid by the locator:
 *   shared data among all tasks.
 */
FAR const char table[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

#pragma restore_attributes
/*
 *   public within task scope: each task can have
 *   it's own instance of the public variable i.
 */
int i;      /* task scope */
/*
 *   static within module scope: each module can have
 *   it's own instance of the static variable s.
 */
static int s;  /* module scope */

TEST_1_NC SECTION LDAT WORD COMMON 'CNEARROM'
_table       LABEL   BYTE
      DB    00h,01h,02h,03h,04h
      DB    05h,06h,07h,08h,09h
      PUBLIC _table
TEST_1_NC  ENDS

TEST_2_NB  SECTION  LDAT WORD PUBLIC 'CNEAR'
TEST_2_NB_ENTRY  LABEL   BYTE
_i    LABEL WORD
      DS    2
      PUBLIC _i
_s    LABEL WORD
      DS    2
TEST_2_NB   ENDS
```

The same object module (containing the common section) must be linked with all tasks using the shared data, because the module name is part of the section name. Of course it is not possible for shared code to access non automatic data which is not shared.

If the medium or large model is used, a shared 'near' data section will cause all near data sections of all tasks to be allocated in the same page, limiting the total near data area of the whole application to 16K. However, it is still possible to have both shared (common) and non-shared (public) near data sections of each task in this area.

If the feature of a 16K near data area for every task is needed, the shared data must be explicitly declared _far (or _huge or _shuge) as done in the example above.

use pragmas 'global' and 'public'

All public declarations in a source file following a pragma 'global' are defined by **c166** at the application (global) scope level in the Task Concept. This means that externs referencing these public variables have to be resolved at the locate stage of **l166**.

Example:

An application consists of two tasks TASK_A and TASK_B.

A module mod_a.c in TASK_A defines a variable which has to be accessed in mod_b.c in TASK_B. The variable (gi) is defined in mod_a.c as follows:

```
#pragma global
unsigned int gi;
#pragma public
```

The #pragma global promotes the scope of the variable gi from the task scope (public) to the application scope (global).

In mod_b.c in TASK_B the variable is declared via:

```
extern unsigned int gi;
```

When linking TASK_B.LNO, the linker will produce a warning about an
'unresolved external _gi'. However, you can tell the linker to check the
unresolved externals with the object file (mod_a.obj) or the task object
file (TASK_A.LNO), which should contain the corresponding global
definition using the CHECKGLOBALS(*object_file*) linker control. If the
corresponding global definition is found by the linker, no warning is
emitted, because the external is resolved at locate time when both TASK_A
and TASK_B are located. The linker and locator invocation may look like:

```
l166 LINK mod_a.obj TO TASK_A.LNO
l166 LINK mod_b.obj TO TASK_B.LNO "CHECKGLOBALS( TASK_A.LNO )"
l166 LOCATE TASK_A.LNO TASK_B.LNO TO tasks.Out
```

define more than one interrupt function in one task

This is the easiest way to share code and data between interrupt functions.
It is in fact a step towards the Flat Interrupt concept. When a task has
more than one entry point (several interrupt functions) reentrancy of the
functions and data must be checked.

use the Flat Interrupt Concept

When the the Flat Interrupt Concept is used, the assembler objects are
directly input for the locator and the linker stage is skipped. The public
(Task) scope level of the Task Concept is promoted to the global
(application) scope level by using the PUBTOGLB (abbreviation PTOG)
locator control. The PTOG control can also be applied to a set of objects
files, which makes it possible to mix the Flat Interrupt Concept with the
Task Concept. When the PTOG is specified for an object file, all public
(task scope) variables and functions are promoted to the application scope
(global) as if they were defined after a pragma 'global'. See the section
l166 Controls in the 'TASKING Cross-Assembler, Linker/Locator, Utilities
User's Manual' for more information about the **l166** linker/locator controls.

LANGUAGE

3.4   DATA TYPES

All (ANSI C) types are supported. In addition to these types, the `_sfr`, `_sfrbit`, `_esfr`, `_esfrbit`, `_bit`, `_xsfr` and `_bitword` types are added. Object size and ranges:

Data Type	Size (bytes)	Range
_bit	1 bit	0 or 1
_sfrbit	1 bit	0 or 1
_esfrbit	1 bit	0 or 1
signed char	1	–128 to +127
unsigned char	1	0 to 255U
_sfr	2	0 to 65535U
_esfr	2	0 to 65535U
_xsfr	2	0 to 65535U
signed short	2	–32768 to +32767
unsigned short	2	0 to 65535U
_bitword	2	0 to 65535U
signed int	2	–32768 to +32767
unsigned int	2	0 to 65535U
signed long	4	–2147483648 to +2147483647
unsigned long	4	0 to 4294967295UL
float	4	+/– 1,176E–38 to +/– 3,402E+38
double	8	+/– 2,225E–308 to +/– 1,797E+308
long double	8	+/– 2,225E–308 to +/– 1,797E+308
_near pointer	2	16 bits (64K) when using –Mt/–Ms 14 bits (16K) when using –Mm/–Ml (default data group)
_xnear pointer	2	14 bits (16K) when using –Mm/–Ml. Not allowed in non–segmented memory models.
_far pointer	4	14 bits (16K) in any page (16M)
_huge pointer	4	24 bits (16M)
_shuge pointer	4	24 bits (16M), but arithmetic is done 16–bit wide

Table 3-12: Data types

- **c166** 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. Section 3.4.2, *Character Arithmetic* provides detailed information.
- The C166/ST10 convention is used, storing variables with the least significant part at low memory address. Float and double are implemented using IEEE single and double precision formats. See section 3.16, *Floating Point Interfacing* in this chapter for more details.

3.4.1   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, may be used in an expression wherever an integer may 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 3.4.2, *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
                          */
}
```

3.4.2   CHARACTER ARITHMETIC

c166 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. This approach increases code density and execution speed (when character typed variables are used of course).

In strict ANSI-C, character arithmetic does not exist: all character variables are converted to integer before the operation is performed.
However, if the integer result is not used (e.g. by assigning it to a character variable) the operation could have been evaluated using character arithmetic, giving the same result. This is how **c166** works.

There is one exception to this rule, dealing with the `sizeof` operator:

```
char a, b;
int  i;

void
main()
{
    i = sizeof( 'A' );      /* -Ac: 1, -AC option: 2 */
    i = sizeof( a + b );    /* -Ac: 1, -AC option: 2 */
}
```

You can enable/disable character arithmetic with the **–Ac**/**–AC** command line option.

3.4.3   THE BIT TYPE

The _bit type is subject to the following rules:

1. A bit type variable is always placed in bit-addressable RAM.

2. A bit type variable is always unsigned.

3. A bit type variable can be exchanged with all other type-variables. The compiler generates the correct conversion.

4. Pointer to a bit-variable and array of bit is **not** allowed, because the C166/ST10 has no instructions to indirectly access a bit variable.

5. 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 as parameter or return value of a function.

6. A union of a bit structure and another type is not allowed. The bitword type can be used for this purpose.

7. A bit type variable is **not** allowed as parameter. The allowed classes for bit are: automatic, static, public or extern.

8. A function may have return type bit.

9. The sizeof of a bit type is 1.

10. A bit typed expression is **not** allowed as switch expression.

The constants need a (bit) cast operator in order to enable bit operations such as '&', '^'. Of course this is not needed with (compound) assignments.

The following table shows which operators are allowed with bit type variables:

Allowed is:

```
==, !=, <, <=, >, >=
&&, ||, !, ~
? :, CALL, RETURN
&, |, ^
&=, |=, ^=
conversions to/from char/int/long/float/double
bit structures (bit members only)
unary plus
```

LANGUAGE

Not allowed or not recommended is:

```
++, --  (post/pre increment/decrement)
unary minus
indirection (array/pointer/address)
+, -, *, /, %, <<, >>
+=, -=, *=, /=, %=, <<=, >>=
```

3.4.4   THE BITWORD TYPE

You can declare word variables in the bit-addressable area as **_bitword**.
You can access individual bits using the intrinsic functions **_getbit()**
and **_putbit()** or declare the individual bits of this **_bitword** variable
using **_atbit**. A prototype for these functions is given in the include file
c166.h.

For example:

```
_bitword bw1, bw2;        /* bitaddressable words */

if ( _getbit( bw1, 3 ) )
     _putbit( 1, bw2, 7 );  /* set bit 7 of bw2 */
```

See also section 3.2.7, *The _atbit() Attribute*.

The **_bitword** type is subject to the following rules.

1. A bitword type variable is always unsigned and volatile.

2. A bitword type variable can be exchanged with all other type-variables.
   The compiler generates the correct conversion.

3. Pointer to a bitword variable and array of bitword is allowed.

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

5. A bitword type variable is not allowed as automatic or parameter. The
   allowed classes for bitword are: static, public or extern.

6. The **sizeof** of a bitword type is same as int.

7. A bitword typed expression is allowed as switch expression.

3.4.5   SPECIAL FUNCTION REGISTERS

c166 recognizes the keywords: **_sfr** and **_sfrbit** to access the special function register area. With the keywords **_esfr** and **_esfrbit** you can access the extended special function reigister area.

c166 also recognizes the keyword: **_xsfr**. The **_xsfr** keyword is used to access special function registers outside the (E)SFR areas but within internal RAM (DPP3). Variables declared as xsfr are not bitaddressble. Example: PEC source and destination pointers (SRCPx/DSTPx).

c166 emits the name of the special function register in the assembly code. **c166** does not perform any check whether the name is correct or not, but passes the name to **a166**. The assembler checks the validity of the name.

For each derivative a special include file *regderivative*.h is delivered with the package, which contains all sfr, xsfr, sfrbit and esfrbit declarations of the selected derivative. Depending on the selected **–x** option, the compiler generates a $STDNAMES assembler control for a default register definition file for the assembler:

–x, **–xd**	$STDNANES(reg.def)
	reg.def contains the C167 register set
–x2	$STDNAMES(regsuper10bo.def)
–x22	$STDNAMES(regsuper10m345.def)

By default **a166** searches files supplied to the STDNAMES control in the `etc` directory installed with the product. This way **a166** finds the file `reg.def` in that directory. To select the same register file for the assembler as for the compiler it is recommended to supply the $STDNAMES control on the command line of the assembler. For example, if your C code includes the file `reg163.h`, you should supply the control `$STDNAMES(reg163.def)` to the assembler on the command line.

All `reg*.h` files consist of a number of parts, which are all included by default. However, if you do not need every part in your source file, you can omit each part by defining the appropriate macro before you include this file. These 'control' macros are described in the `reg*.h` files.

REG163_NOPORT	omit port I/O registers
REG163_NORS232	omit serial I/O registers
REG163_NOTIMER	omit timer registers
REG163_NOADINT	omit additional peripheral
REG163_NOEXTINT	omit fast external interrupt

REG165_NOCPU	omit cpu registers
REG165_NOPEC	omit PEC registers
REG165_NOPORT	omit port I/O registers
REG165_NORS232	omit serial I/O registers
REG165_NOTIMER	omit timer registers
REG165_NOADINT	omit additional peripheral interrupt registers
REG165_NOEXTINT	omit fast external interrupt registers
REG166_NOADC	omit analog/digital registers
REG166_NOCAPCOM	omit capture/compare registers
REG166_NOCPU	omit cpu registers
REG166_NOPEC	omit PEC registers
REG166_NOPORT	omit port I/O registers
REG166_NORS232	omit serial I/O registers
REG166_NOTIMER	omit timer registers
REG167_NOADC	omit analog/digital registers
REG167_NOCAPCOM	omit capture/compare registers
REG167_NOCPU	omit cpu registers
REG167_NOPEC	omit PEC registers
REG167_NOPORT	omit port I/O registers
REG167_NORS232	omit serial I/O registers

You can make your own special function register header file, but in that case you must supply the same names to **a166** by an STDNAMES file.

c166 and **a166** do not generate symbolic debugging information for special function registers, because the register names should be known by the debugger.

Because the special function registers are dealing with I/O, it is not correct to optimize away the access to these registers. Therefore, **c166** deals with special function registers as if they were declared with the volatile qualifier.

```
_sfr    var1; is treated like: volatile unsigned int var1;
_sfrbit var2; is treated like: volatile _bit var2;
_xsfr   var3; is treated like: volatile unsigned int var3;
```

3.5 PREDEFINED MACROS

In addition to the predefined macros required by the ANSI C standard, the TASKING C compiler supports the predefined macros as defined in Table 3-13. The macros are useful to create conditional C code.

Macro	Description
_DOUBLE_FP	Defined when you do not use compiler option **–F** (Treat double as float)
_SINGLE_FP	Defined when you use compiler option **–F** (Treat double as float)
_C166	Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the **c166** compiler only. It expands to the version number of the compiler. For example, if you use version 8.6r3 of the compiler, _C166 expands to 86 (dot and revision number are omitted).
_CPUTYPE	Expands to a value representing the CPU type, depending on option **–x**: **–x** 0x167 (default) **–xd** 0x272 **–x1** 0x1661 **–x2** 0x1662 **–x22** 0x16622
_MODEL	Identifies the memory model. Expands to the argument of option **–M**. See section 3.2.1.6, _MODEL.
_USMLIB	Expands to _usm if **–P** is specified, or _nousm otherwise. See section 3.2.2, *User Stack Model*, for more information.

Table 3-13: Predefined macros

Example:

```
#if _CPUTYPE == '0x1662'  /* XC16x/Super10 */
...
#endif
```

3.6 FUNCTION PARAMETERS

A lot of execution time of an application is spent transferring parameters between functions. Therefore this is an area which is very interesting for optimization. The conventional CPU approach for parameter passing is via the stack, because C allows recursion and reentrancy (the stack sizes of each task are accumulated by the locator stage of **l166**).

Because it is very important to optimize parameter passing, **c166** uses a resource which a RISC processor like the C166/ST10 has plenty of: registers. The first parameters are placed in specific registers (R12- R15). Very often the parameter computation can be done directly in the appropriate register. In practice the bulk (80-90%) of the calls pass four or fewer (word-sized) parameters.

A special keyword `_stackparm` is introduced as a 'function qualifier' (like `_interrupt`) to tell the code generator to pass all parameters via the user stack. This keyword is very convenient for interfacing with (existing) assembly functions or when register usage must be minimized (e.g. **–r6** is used for a small C interrupt function calling another C function):

```
void _stackparm assembly_function( char type,
                                   long size );
```

Register parameter passing is NOT done if one of the following conditions is true:

- the 'dot arguments' of a function having a variable argument list (ANSI notation of prototype declaration, using three dots, e.g.: `void f( char *, ... );`)
- the called function has a prototype with the `stackparm` function qualifier.
- the register parameters are already full or one of the parameters cannot be passed in a register (explained below in more detail).

If a variable argument list function (e.g. `printf()`) is called without a valid prototype (`#include <stdio.h>`) run-time errors occur due to parameter **transfer** mismatches.

If a function prototype is used with a function call but NOT with the function body (or vice versa), run-time errors may occur due to parameter **type** mismatches.

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 saved. Thus, the parameters of such a function are left in the input registers. A lot of C library functions (such as strlen(), strcpy() etc.) meet these requirements.

Non-leaf functions must save the parameter registers on the user stack at function entry, as if they were pushed by the caller. However, the code generator tries to use the register copies of these parameters as long as possible. If automatic registers are available, these registers are used instead of the user stack.

If a parameter does not fit (anymore) in the parameter registers or the parameter is a float/double or a structure/union (not a pointer), it is passed via the (more conventional) user stack. All next parameters are passed via the stack to maintain correct stack offsets, even if one of these next parameters would fit in the register area. The following examples (small model) clarify this item:

Example 1:

```
void func1( long l1, int i, long l2, char *p );
/*           R12-R13  R14    stack    stack: not R15 */
```

better:

```
void func1( long l1, int i, char *p, long l2 );
/*           R12-R13  R14    R15      stack  */
```

Example 2:

```
void func2( double d, double *p, int i );
/*          stack     stack      stack  */
```

better:

```
void func2( double *p, int i, double d );
/*          R12        R13     stack  */
```

3.6.1   STATIC APPROACH OF FUNCTION AUTOMATICS

Function automatics (not parameters) which can not be allocated to a register are present on the user stack. Compared to static variables these stack variables have the following disadvantages:

- Access to these variables is only possible via an 'indirect register plus offset' addressing mode. This addressing mode is supported in the following two instructions only:

  1) MOV Rn,[Rm+#d16]
  2) MOV [Rm+#d16],Rn

  This means that all arithmetic operations (add, and, cmp, or, subb and xor) with a stack variable need an extra register move, before the operation can be done. With static memory variables a register move is not needed, because the operations mentioned above allow the usage of the MEM operand.

- Heavy usage of instruction 1) is slowing down execution time, because this instruction takes twice as much time as any other move instruction or arithmetic operation (200ns instead of 100ns at 40MHz).

Therefore, code size and execution speed can be improved if the non–register function automatics may be treated by the compiler as if they were static and it is possible to allocate these 'automatic' variables in the fast internal RAM of the C166/ST10 using a CLASSES or ADDRESSES(SECTIONS) locator control. Of course, this is not possible with recursive functions. Because function automatics do not have any interaction with other functions (unlike parameters), it is not necessary to introduce a special static model to support this optimization. It is even possible to enable this optimization for only one function in a module.

The compiler supports two ways of specifying function automatics can be treated in a static way:

1. command line option.

   –S   All functions of the C module are compiled using static memory for non register function automatics. This option may be useful for non recursive applications.

2. pragmas.

   If only a few functions of the entire application are recursive, the following pragmas can be used to enable (or disable) this optimization:

   pragma static   Use static memory for non register function automatics.

pragma automatic   Default (unless **–S** is used). Use stack approach
                                for non register function automatics. Support
                                recursion.

The usage of the **–S** option (or **pragma static**) does not change the
semantic behavior of **c166** with automatics: explicit storage type specifiers
(far, near, huge, shuge) remain illegal and the initialization of an automatic
variable is done run-time (each time the function is entered).

3.7   REGISTER VARIABLES

Via the `register` keyword you are able to control which automatic
variable must be allocated to a CPU register by the code generator.
However, if the `register` keyword is NOT used, the front end phase of
c166 determines which C automatic variables might be allocated to a
register by the code-generator (unless the **–OR** option is specified to turn
this optimization off).

If a C function is a non-leaf function (i.e. calling another C function), four
registers (R6-R9) are available to support C register variables. However, if
the C function is a leaf function, not occupied registers of the parameter
register area (R12-R15) can be used for automatic registers too. These
registers do not have to be saved at entry and restored at exit. Thus, leaf
functions allow up to eight registers to be used for register automatics!

The code generator of **c166** uses a 'saved by callee' strategy. This means
that a function which needs one or more registers for register variables,
must save the contents of these registers and restore before returning to
the caller. The major advantage of this approach is, that only registers
which are really used by the function are saved. If the function does not
have any register variable, the registers of the caller function remain valid
without being saved.

The code generator prefers to assign the register character type automatics
to R6 or R7 (using RL6/RL7) and the other types to the rest in the order of
their declaration.

A declaration like (`f()` being a non-leaf function):

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

would have been allocated by the code generator in the following registers:

```
i   ==>   R9
c   ==>   RL6
l   ==>   R7-R8
```

If `f()` would have been a leaf function, the register automatics would have been allocated in the following registers:

```
i   ==>   R15
c   ==>   R14
l   ==>   R12-R13
```

All basic data types which are allowed as automatic variable are supported, except float/double/bit: char, int, long, near/far/huge/shuge pointer. Of course `_sfr`, `_sfrbit`, `_xsfr` and `_bitword` are not possible.

If register usage must be minimized (e.g. interrupt function/module), specify **–r6** on the command line (R0-R5 used in REGDEF). When the **–r** option is used, the automatic register allocation scheme of **c166** is adjusted to meet the requirements of the user.

3.8 INITIALIZED VARIABLES

There are two types of initialized variables, which depend on the class of the variable: `static` or `automatic`. The implementation is described in the following sections.

3.8.1 AUTOMATIC INITIALIZATIONS

Automatic initialized variables are initialized (run-time) each time a C function is entered. Normally, this is done by generating code which assigns the value to the automatic variable.

In the old (K & R) language definition it was not allowed to initialize an automatic aggregate type (e.g. an array or structure), but only integral types. The ANSI standard also allows run-time initialization of automatic aggregate types. To support this feature, **c166** generates code to copy the initialization constants from ROM to RAM each time the function is entered.

3.8.2 STATIC INITIALIZATIONS

There is a lot of existing C source which use static initializations. Static initialized variables normally use the same amount of space in both ROM and RAM. This is because the initializers are stored in ROM and copied to RAM at start-up. In the task philosophy of **c166**, this ROM to RAM copy has to be performed at 'startup' for each task.

c166 takes care of a mechanism, which is completely transparent for the user. It performs initialization per task from system startup code, using compiler generated tables.

Static initialized variables use the same amount of space in both ROM and RAM. The only exception is an initialized variable residing in ROM, by means of either the `#pragma romdata` or the `const` storage type qualifier. For normal initialized RAM variables, you can specify the class name ('CINITIRAM' or 'CINITERAM') to be used with `#pragma iramdata` or `#pragma eramdata`. You can use the CLASSES locator control to affect the location of these variables. See section 3.2.3, *Section Allocation*, for details on section names and section attributes.

Example *(using small model):*

```
const char b = 'b';     /* 1 byte  in ROM */
#pragma iramdata        /* default, may be omitted, unless pragma
                           romdata/eramdata was used before */
int  i = 100;           /* 2 bytes in ROM, 2 bytes in IRAM */
char a = 'a';           /* 1 byte  in ROM, 1 byte  in IRAM */
char *p = "ABCD";       /* 5 bytes in ROM (for "ABCD") */
                        /* 2 bytes in ROM, 2 bytes in IRAM
                           (for p)*/

#pragma romdata         /* Needed for ROM only allocation */
int  j = 100;           /* 2 bytes in ROM */
char *q = "WXYZ";       /* 5 bytes in ROM (for "WXYZ") */
                        /* 2 bytes in ROM (for p) */
```

c166 treats <u>romdata</u> variables as if they were declared with the `const` storage type qualifier.

3.9   NON-INITIALIZED VARIABLES

In some cases there is a need to keep variables unchanged even if power is turned off. In these systems some of the RAM is implemented in EEPROM or in a battery-powered memory device. In a simulator environment, clearing non-initialized variables might not be wanted too.

To avoid the 'clearing' of non-initialized variables at startup, one of the following things should be performed:

1. Define (allocate) these variables in a special C module and compile this module using the **–Ob** option. **c166** will omit these data sections, when building the C166_BSS section.

From EDE: from the **Projects** menu, select **Project Options...** Expand the **C Compiler** entry and select **Allocation of Variables**. Disable the check box **Perform 'clearing' of non–initialized static/public variables**.

2. Define (allocate) these variables between **#pragma noclear** and **#pragma clear**. **c166** will omit these data sections, when building the C166_BSS section.

The last **#pragma** [**no**]**clear** before or in a function, applies to all static/global variables, in or outside a function.

3. Use inline assembly to allocate the special variables in a special data section (NOT used by other C variables).

4. Make a separate assembly module, containing the allocation of these variables in a special data section.

It is not possible to remove the 'clearing code' from the startup file, because other C modules (and the C libraries) depend on it too.

Variables in bit-addressable RAM are cleared by default and not effected by any of the above mentioned methods. However, you can disable this automatic clearing from EDE: from the **Projects** menu, select **Project Options...** Expand the **Application** entry and select **Startup**. Disable the check box **Clear bit–addressable RAM at startup**.

Because variables in bit-addressable memory are cleared by the startup code, the compiler does not allocate structs with bit-fields only, into bit-addressable memory automatically. You can tell the compiler that a struct must be located in bit-addressable memory by using the _bita memory qualifier. See the _bita keyword in section 3.2.1.9, *_system, _iram and _bita*.

3.10 STRINGS

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

Strings have static storage. The ANSI X3.159-1989 standard permits string literals to be put in ROM. Because there is no difference in accessing ROM or RAM, **c166** allocates strings in ROM only. This approach also saves RAM, which can be very scarce in an embedded (single chip) application.

As mentioned before, **c166** offers the possibility to allocate a static initialized variable in ROM only, when declared with the const qualifier or after a #pragma romdata. This enables the initialization of a (const) *character array* in ROM:

```
const char romhelp[] = "help";
/* allocation of 5 bytes in ROM only */
```

Or a pointer array in ROM only, initialized with the addresses of strings, also in ROM only:

```
char * const messages[] = {"hello","alarm","exit"};
```

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. To save ROM space, **c166** overlays identical strings within the same module.

Allocation of string constants

By default the compiler allocates string constants in the memory model's default memory space. You can overrule this with #pragma stringmem:

> **#pragma stringmem** *memory-space*

Where *memory-space* is one of:

Memory Space	String Location
_near	near ROM
_xnear	xnear ROM
_far	far ROM
_shuge	shuge ROM
_huge	huge ROM
default	memory model default

The 'default' argument allocates strings in the memory model's default memory space. In the small memory model this also means that the **–Oe/–OE** option is effective. See also section 3.2.5 *Constant Romdata Section Allocation*.

The following example illustrates the use of the pragma:

```
#pragma stringmem _huge  /* allocate strings in _huge memory */
_huge char * txt = "text1";
```

• • • • • • • • •

This results in the following code:

```
STR_IR_NB        SECTION PDAT WORD PUBLIC 'CINITROM'
STR_IR_NB_ENTRY LABEL    BYTE
        DSPTR    _3
STR_IR_NB        ENDS


STR_ID_NB        SECTION LDAT WORD PUBLIC 'CINITIRAM'
STR_ID_NB_ENTRY LABEL    BYTE
_txt    LABEL    WORD
        DS       4
        PUBLIC   _txt
STR_ID_NB        ENDS


STR_3_HC         SECTION HDAT WORD PUBLIC 'CHUGEROM'
_3      DB       074h,065h,078h,074h,031h,00h
STR_3_HC         ENDS


C166_INIT        SECTION PDAT WORD GLOBAL 'CINITROM'
        DW       06h
        DPPTR    STR_ID_NB_ENTRY,STR_IR_NB_ENTRY
        DW       04h
C166_INIT        ENDS
```

The pragma can appear anywhere in the source and remains in effect until
the pragma is used again to set a different memory space.

3.11 INLINE ASSEMBLY

c166 supports an inline assembly facility by means of the following pragmas:

#pragma 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.
#pragma asm_noflush	Same as **asm**, except that the peephole optimizer does not flush the code buffer and assumes register contents remain valid.
#pragma endasm	Switch back to the C language.

You should realize that using these pragmas results into non portable and hard to 'simulate' code. Therefore, usage of these pragmas should be minimal.

C Variable Interface for Pragma asm

The pragma **asm** and **endasm** synopsis of the pragmas is as follows:

#pragma asm [(*pseudo_reg*[=*varname*][**,** *pseudo_reg*[=*varname*]] ...**)**]

#pragma endasm [(*varname*=*pseudo_reg*[**,** *varname*=*pseudo_reg*] ...**)**]

The arguments of the pragmas are:

varname   name of a C variable of type char or int, signed or unsigned,

pseudo_reg   a pseudo register name with the synopsis:

@[**w**|**b**|**i**]*num*

w	word register R0–R15
b	byte register RL0-7, RH0-7
i	indirect address register R0-R3, some addressing modes only support these registers

num   a user defined number of the pseudo register. This
number is not related to the register that is substituted
by the compiler. The number must be in the range
0-15.

When no **w**, **b**, or **i** is given a word register is used.

Examples:

@1   word register pseudo

@w2   word register pseudo

@b3   byte register pseudo

@i4   word register pseudo

When a *pseudo_reg* is listed without assignment of a *varname*, the
compiler will reserve a scratch register. When in the pragma **endasm** a
pseudo_reg is listed that is not listed in the pragma **asm**, it will also be
reserved as a scratch register.

Example:

```
#pragma asm( @w1=var1, @b2=var2, @i3=var3, @4  )
     EXTERN XVAL:WORD, BVAL:BYTE, YVAL:WORD
     MOV @4, @w1      ; fill temporary register
     MOV XVAL, @4     ; save in some memory location
     MOV BVAL, @b2    ; save in some memory location
     MOV @i3, #2      ; small instruction (Rn, #data4)
     MOV @w1, YVAL    ; get some memory location
#pragma endasm( retval=@w1 )
```

The compiler will take care that the requested registers are free to be used
and that their original content is saved and restored if needed. When the
compiler is not capable of allocating registers for the listed pseudos an
error message will be issued. The number of pseudos that can be allocated
for inline assembly depend on the complexity and size of the C code part
of the function.

Defining inline assembly functions can be done by using the pragma asm
interface in an inline C function.

See section 3.17.1, *User Defined Intrinsics* in this chapter.

Example:

```
_inline int swap_add( int a, int b )
{
      int rv;
#pragma asm ( @1=a, @2=b, @3 )
      MOV  @3, @1
      MOV  @1, @2
      MOV  @2, @3
      ADD  @3, @1
#pragma endasm ( rv=@3 )
      return rv;
}
```

Known restriction: The #pragma asm may cause an inline assembly to be optimized away by the **c166** flow optimizations. For example:

```
void example(void)
{
   goto the_end;
#pragma asm
entry:
  ; assembly statements here will not be emitted by c166
  ; because it is considered ''not reachable'', even when
  ; the assembly starts with a label.
#pragma endasm
the_end: ;
}
```

Workaround for this restriction: Replace C statements which seems to make the inline assembly not reachable by an assembly equivalent inside the #pragma asm:

```
void workaround(void)
{
#pragma asm
   jmp     the_end
entry:
   ; assembly statements here will be emitted by c166
the_end:
#pragma endasm
}
```

See also section 7.4, *Assembly Language Interfacing* in the chapter *Run-time Environment*.

The 'MODULE SUMMARY' of **c166**, reporting code size and data size of the module, is no longer valid if code or data has been added using inline assembly.

3.12 INTERRUPT

c166 supports both the 'Infineon Task Concept' and the 'Flat Interrupt Concept'. These two concepts are explained in the chapter *Software Concept* of the 'TASKING Cross–Assembler, Linker/Locator, Utilities User's Manual'. We strongly recommend reading this section first! See also section 3.3 *Task Scope* in this chapter.

In the Task Concept a Task is initiated via an interrupt or software trap. The 'reset task' is the task which defines main. The system startup file ('start.asm' in assembly code) delivered with the compiler, initializes the processor and each task and finally calls `main()`. In the Flat Interrupt concept an interrupt is an entry point in the code. The system startup code is such an entry point.

Defining an Interrupt Function: _interrupt()

With the function type qualifier `_interrupt()` you can declare a function as an interrupt function. Interrupt functions cannot return anything and must have a void argument type list. The function type qualifier `_interrupt()` takes one interrupt number (-1, 0..127) as argument:

```
void _interrupt( interrupt_number )
isr( void )
{
    ...
}
```

For example:

```
void _interrupt( 0x22 ) timer( void )
{
    ...
}
```

The interrupt number -1 is reserved for a so-called symbolic interrupt. This means that **c166** does not assign an interrupt number to this C function. The interrupt function can be bound to any interrupt number in the locate stage of **l166** by the INTERRUPT control.

Interrupt Frame

c166 generates an interrupt frame inheriting the user stack pointer from the previous task, switching context to a new register bank, saving DPP registers and MDC, MDH and MDL registers. When the **–Oh** command line option is set (default) the compiler optimizes the interrupt frame so that it only contains the parts needed to save resources used by the interrupt function. You can also tell the compiler to omit the whole interrupt frame via the following pragma:

```
#pragma noframe
```

This allows you to make your own interrupt frame.

With the _using(push) function qualifier you can tell the compiler to use PUSH/POP instructions to save GPRs in the interrupt frame, instead of using a SCXT instruction. Only the GPRs that are used in the interrupt function, are saved.

```
void _interrupt( 0x22 ) _using(push) timer( void )
{
    ...
}
```

With the _using(push) function qualifier, it is allowed to use the _localbank() qualifier. The compiler will use the selected localbank only to administrate the user stack size estimates on the correct stack. See section 3.13 *Extensions for the XC16x/Super10 Architectures* for more information.

Register Banks: _using()

When you use the _using() function qualifier with a register bank name as an argument, you can tell the compiler to generate a new register bank for the interrupt function. For example:

```
_interrupt( 0x28 ) _using( ADCONV_RB ) void
ad_conv_complete( void )
{
    ...
}
```

This way you can define several interrupt functions in one module with each function having its own register bank. Or you can share a register bank between several interrupt functions which have the same interrupt level and thus can never interrupt each other. When several interrupt functions in a source module are 'using' a register bank with the same name, the compiler uses the same register bank for these functions. **1166** will 'overlay' register banks with equal names.

All interrupt functions without the _using keyword use a register bank with a name derived from the module name. This means that all interrupt functions in one C source file which do not have the _using keyword use the same register bank and therefore they should have the same interrupt level. Different interrupt levels can be used, but in this case #pragma regdef is needed to instruct the compiler to use non-overlapping register sets.

With #pragma regdef you can define the register set that the compiler uses for code generation. The pragma affects all functions after the pragma, until #pragma regdef is used again to define another register set. When #pragma regdef is used without an argument, or with argument 0, the REGDEF assembler directives used for interrupt functions will be omitted, even when the _using() qualifier is used. In this case the compiler does not generate code to switch to another global register bank.

You can use the **-r** command line option to name the register bank of a module. With an optional flag the register bank can be declared 'common'. When a register bank definition is supplied with the **-r** option, this register set is used until the next #pragma regdef in the source.

When the **-r** option is used without any arguments, the REGDEF directive for this module will be omitted. This does not affect the REGDEF directives originating from the _using() qualifier. Interrupt functions that do not have the _using() qualifier use the module's REGDEF. Since this REGDEF is omitted, no code will be generated in the interrupt frame to switch register banks.

See the description of the **-r** option for more details.

LANGUAGE

3.13 EXTENSIONS FOR THE XC16x/Super10 ARCHITECTURES

The XC16x/Super10 architectures support fast register bank switching using local register banks. You can make use of this feature using the `_localbank` keyword. This keyword can only be applied on interrupt functions.

```
_localbank (num)
```

Where `num` can be one of the following:

-3:   Use local register bank 3 but assume the hardware automatically swithches the register bank upon interrupt. (Enhanced Super10 only, option **–x22**)

-2:   Use local register bank 2 but assume the hardware automatically swithches the register bank upon interrupt.

-1:   Use local register bank 1 but assume the hardware automatically swithches the register bank upon interrupt.

0:          Use global register bank as usual.

1:          Use local register bank 1 and emit instruction in interrupt frame to select the correct local register bank.

2:          Use local register bank 2 and emit instruction in interrupt frame to select the correct local register bank.

3:          Use local register bank 3 and emit instruction in interrupt frame to select the correct local register bank. (Enhanced Super10 only, option **–x22**)

Only the `_localbank (0)` qualifier can be used in conjunction with the `_using` qualifier. The correct register bank will not be selected when `#pragma noframe` is entered before the interrupt function.

Since local register banks are not memory mapped, the compiler can not copy the user stack pointer `(R0)` to the new register bank. Therefore, each local register bank will have its own user stack area:

```
C166_US0: will be used together with register bank 1

C166_US1: will be used together with register bank 2
```

The compiler estimates the size of each seperate stack based upon the code inside interrupt functions only. User stack space occupied by functions which are called from the interrupt function are not taken into account.

The estimated user stack size can be adjusted using a new function qualifier:

```
_stacksize (num)
```

Where num specifies the user stack adjustment in bytes. A positive number increases the compiler estimates by num bytes, a negative value decreases it. If the sum of the compiler estimation and the stack adjustment is negative, a warning will be generated and the value will be truncated. The value of num must be even.

The _stacksize qualifier can only be used in combination with the local register banks (for example: _localbank (0) is NOT allowed) and interrupt functions.

User stack size estimations will not be performed if #pragma nocustack was used. Of course it is still possible to adjust the size of the generated user stack sections at locate time using the SECSIZE control.

The complete definition of an interrupt function could look like this:

```
/*
 * Define an interrupt function using local register
 * bank 1 assuming the hardware automatically selects
 * local bank 1 upon interrupt. Increase the compiler
 * estimated user stack size by 40 bytes. The user
 * stack will be allocated in section: C166_US0
 */
void _interrupt(0x10) _localbank(-1) _stacksize(+40)
ISR(void)
{
    return;
}
```

Another feature of the ext2 architectures is the *scalable interrupt vector table*. The compiler uses this feature by trying to inline as much code as possible inside the interrupt vector table. Small interrupt functions can be located inside the vector table completely. This will improve interrupt latency. The size of an entry in the interrrupt vector table can be supplied to the compiler by the command line option:

-i*num*

Where *num* can be one of the following:

```
0 - No scaling          (4 bytes/entry)
1 - 2x the normal size (8 bytes/entry)
2 - 4x the normal size (16 bytes/entry)
3 - 8x the normal size (32 bytes/entry)
```

When either option is supplied to the compiler, it will try to reorder and move code from the interrupt frame to the interrupt vector table. Where possible the context switch will be done just before the JMPS instruction which jumps to the ISR. By doing this, the execution time of the JMPS instruction will be hidden by the context switch.

the compiler will put all sections that have to be inlined in a special section called:"C166_INT" with class:"C166_VECTAB". An example of an inlined interrupt function is shown below:

```
; **************************************************
; * Section which will be located at vector position
; * 0x10 by the locator, the scaling = 3
; * (32bytes/entry available in vector table)
; **************************************************

C166_INT   SECTION CODE WORD PUBLIC 'C166_VECTAB'
_3    PROC TASK SCALEDVE_TASK INTNO
      SCALEDVE_INUM = 010h SCALING 3 INLINE

      PUSH CP                            ;; 2 bytes
      SCXT MDC,#010h                     ;; 4 bytes
      PUSH DPP0                          ;; 2 bytes
      MOV  DPP0,#PAG ?BASE_DPP0          ;; 4 bytes
      PUSH DPP2                          ;; 2 bytes
      MOV  DPP2,#PAG ?BASE_DPP0          ;; 4 bytes
      PUSH MDH                           ;; 2 bytes
      MOV  SCALEDVE_RB,R0                ;; 4 bytes
      MOV  CP,#SCALEDVE_RB               ;; 4 bytes
           ;; (Context switch right before JMPS)
      JMPS  SEG _ISR1,_ISR1             ;; 4 bytes
      RETV                               ;; --------+
_3   ENDP                                ;; 32 bytes
C166_INT ENDS
```

```
; ********************
; * Start of ISR
; ********************

      SCALEDVE_1_PR SECTION CODE
_ISR1 PROC TASK ISR
      PUSH MDL

; ********************
; * User code goes here
; ********************

              POP    MDL
              POP    MDH
              POP    DPP2
              POP    DPP0
              POP    MDC
              POP    CP
              RETI
_ISR1         ENDP
```

A faster way to trasfer control to an interrupt function is to make use of
cached interrupts. To support this, the hardware of the ext2 architectures
bypasses the interrupt vector table at all. In this case, the compiler cannot
inline any code of the interrupt fuction in the vector table. Therefore, the
_cached keyword has to be used on these interrupt functions. The
following code fragment gives an example of the use of the _cached
function qualifier:

```
void  _interrupt (0x10) _localbank(-1) _cached
      ISR(void)
{
      return;
}
```

The _cached function qualifier will basically overrule the **–i** command
line option causing none of the code to be located inside the interrupt
vector table.

LANGUAGE

Examples:

1. The C module is called 'intrpt.c' (present in the **examples/c** directory). The example illustrates how to tell the compiler to omit the interrupt frame code. The C source and the generated code (large) is listed below:

```
#pragma global
bit b;        /* interrupt handler sets a global bitvariable */
#pragma public

#pragma noframe          /* minimal interrupt frame     */
                         /* even no GPR's needed, so     */
#pragma regdef 0         /* omit regdef definition       */

interrupt (0x30) void
f()
{
#pragma asm
     NOP                 ; you can make your own entry code here
#pragma endasm
     b = 1;
#pragma asm
     NOP                 ; you can make your own exit code here
#pragma endasm
}


INTRPT_1_BI  SECTION  BIT BIT PUBLIC 'CBITS'
INTRPT_1_BI_ENTRY  LABEL  BIT
_b  DBIT
    GLOBAL    _b
INTRPT_1_BI  ENDS

INTRPT_2_PR  SECTION  CODE WORD PUBLIC 'CPROGRAM'
_f    PROC TASK INTRPT_TASK INTNO INTRPT_INUM = 030h
      NOP                ; you can make your own entry code here
      BSET _b
      NOP                ; you can make your own exit code here
      BCLR IEN
      RETI
_f    ENDP
INTRPT_2_PR  ENDS
```

2. The C module is called 'intrpt.c' (present in the **examples** directory). The example illustrates the use of '#pragma regdef' and shows the code the compiler emits as interrupt frame using large memory model (DPP0 and DPP2 saving). The user stack pointer must be inherited and the multiply registers must be saved. The C source and the generated code is listed below:

• • • • • • • • •

```
#pragma regdef 6          /* MINIMIZE REGISTER USAGE to R0-R5 */

int stackparm ext_func( int );  /* stack parameter passing: NOT
R12-R15 */

interrupt (0x30) void
f()
{
      int i;              /* allocate on user stack: NOT R6-R9 */

      i = ext_func( 3 );
}


INTRPT_1_PR  SECTION  CODE WORD PUBLIC 'CPROGRAM'
_f    PROC TASK INTRPT_TASK INTNO INTRPT_INUM = 030h
; Stack: 2
      MOV    DPP3:INTRPT_RB,R0
      SCXT   CP,#DPP3:INTRPT_RB
      SCXT   MDC,#00h
      PUSH   DPP0
      PUSH   DPP2
      MOV    DPP2,#PAG C166_DGROUP
      PUSH   MDL
      PUSH   MDH
      SUB    R0,#02h
      MOV    R4,#03h
      MOV    [-R0],R4
      CALLS  SEG _ext_func,_ext_func
      ADD    R0,#02h
      MOV    [R0],R4
      ADD    R0,#02h
      POP    MDH
      POP    MDL
      POP    DPP2
      POP    DPP0
      POP    MDC
      POP    CP
      BCLR   IEN
      RETI
_f    ENDP
INTRPT_1_PR  ENDS

INTRPT_RB    REGDEF  R0-R5
```

Instead of using #pragma regdef 6 you can also use the command line
option **-r6**. When you use the **-r** command line option, you can also
specify the register bank name to be used and whether this register bank
should be COMMON or not.

Specifying **-r6,MYBANK,c** results into:

```
MYBANK  REGDEF R0-R5 COMMON = MYBANK_RB
```

It is very useful to share the register bank of interrupt functions, which are at the same interrupt priority level, so they cannot be active simultaneously. This approach saves internal RAM space, which is a scarce resource.

3. The C module is called 'intrpt.c' (present in the **examples** directory). The examples illustrates the **using** keyword. The C code and the generated code (large memory model) is listed below:

```
int i;

interrupt (0x30) using (INTRPT_RB) void
f()
{
    i+=2;
}

        ASSUME      DPP3:SYSTEM
INTRPT_1_NB         SECTION  DATA WORD PUBLIC 'CNEAR'
        ASSUME      DPP2:INTRPT_1_NB
INTRPT_1_NB_ENTRY LABEL BYTE
_i      LABEL WORD
        DS      2
        PUBLIC _i
INTRPT_1_NB  ENDS

INTRPT_2_PR  SECTION   CODE WORD PUBLIC 'CPROGRAM'
_f    PROC TASK INTRPT_TASK INTNO INTRPT_INUM = 030h
; Stack: 0
        MOV   DPP3:INTRPT_RB,R0
        SCXT  CP,#DPP3:INTRPT_RB
        PUSH  DPP2
        MOV   DPP2,#PAG C166_DGROUP
        MOV   R4,#02h
        ADD   _i,R4
        POP   DPP2
        POP   CP
        BCLR  IEN
        RETI
_f    ENDP
INTRPT_2_PR  ENDS

C166_BSS    SECTION  DATA WORD GLOBAL 'CINITROM'
        DW    06h
        DPPTR INTRPT_1_NB_ENTRY
        DW    02h
C166_BSS    ENDS

C166_DGROUP DGROUP      INTRPT_1_NB
INTRPT_RB   REGDEF      R0-R15
        REGDEF      R0-R15
        END
```

• • • • • • • • •

3.14 SWITCH STATEMENT

c166 supports two ways of code generation for a switch statement: a jump chain or a jump table. A jump chain is comparable with an if/else-if/else-if/else construction. If all of the following conditions are true, a jump table is emitted:

1. type is not long (char, int, bitfield only)

2. at least five case labels are present

3. total number of 'gaps' between the case labels (when sorted) does not exceed the number of case labels.

It is obvious (especially with large switch statements) that the jump table approach executes faster than the jump chain approach. If speed is needed (e.g. an interrupt function) it might be acceptable to use a jump table, even if the number of gaps between the (sorted) case labels exceeds the number of case labels itself. Therefore the second and third requirement can be overruled by using:

```
#pragma switch_force_table
```

and restored using:

```
#pragma switch_smart
```

which is the default situation. The command line equivalents are **–Os** (switch_force_table) and **–OS** (default, switch_smart).

The location of jump tables in the small memory model can be controlled by using

```
#pragma switch_tabmem_far
```

which places jump tables in class 'CFARROM'.

```
#pragma switch_tabmem_near
```

which places jump tables in class 'CNEARROM'.

```
#pragma switch_tabmem_default
```

which places jump tables on the default location, which is controlled by the **–Oe**/**–OE** command line option. This is the default.

See section 3.2.5 *Constant Romdata Section Allocation* for details.

3.15 REGISTER USAGE

c166 uses the general purpose registers (GPRs) of the C166/ST10 as follows:

Register	Usage
R0	User Stack Pointer (USP)
R1–R5, R10, R11	General registers (codegen, temporary results, C return values)
R6–R9	C register variables and saved register parameters
R12–R15	Fast C parameter passing and C register variables

Table 3-14: General purpose registers

c166 uses the following registers for C function return types:

Return type	Register(s)
bit	PSW.6 (USR0)
char	RL4
short/int	R4
long	R4–R5 (R4 low word, R5 high word)
float	R4–R5
double	user stack and R4
structure	user stack and R4 (pointer to stack block)
near pointer	R4
far pointer	R4–R5 (R4 page offset, R5 page number)
huge pointer	R4–R5 (R4 segment offset, R5 segment number)
shuge pointer	R4–R5 (R4 segment offset, R5 segment number)

Table 3-15: Register usage for function return types

3.16 FLOATING POINT INTERFACING

3.16.1 INTRODUCTION SOFTWARE FLOATING POINT USAGE

Section 3.16 describes the usage of floating point numbers. This includes storage format, trap handling and usage in assembly programs.

3.16.2 THE IEEE-754 FORMAT

Floating point numbers are stored in IEEE-754 format. This manual explains its format only briefly. For a more detailed version you are referred to the IEEE-754 standard, published by the Institute of Electrical and Electronic Engineers, Inc.

Basic single precision format

The basic single precision format is like this:

seeeeeeeemmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa	

You can convert this to an understandable number with the formula:

$$value = (-1)^s \cdot \left(1 + \frac{m}{2^{23}}\right) \cdot 2^{e-127}$$

An example:

```
0x40490fdb
s = 0
e = 0x80 = 128
m = 0x490fdb = 4788187
```

$$value = (-1)^0 \cdot \left(1 + \frac{4788187}{8388608}\right) \cdot 2^1 = 1 \cdot (1 + 0.5707964) \cdot 2 = 3.14159274$$

Special case single precision 0.0

0.0 is stored as:

s000000000000000	0000000000000000
seeeeeeeemmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa	

Notice that there is a +0.0 and a -0.0.

Special case single precision NaN (Not a Number)

Generated when the result of an expression is undefined e.g. 0.0 / 0.0.

NaN is stored as:

s111111111111111	1111111111111111
seeeeeeeeemmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa	

According to the IEEE standard not all mantissa bits have to be set for a number to be handled as NaN.

Special case single precision INF (Infinity)

Generated when the result of an expression is larger than can be stored, e.g. 1.0e30f * 1.0e30f.

INF is stored as:

s111111110000000	0000000000000000
seeeeeeeeemmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa	

Sign defines +INF or -INF.

Basic double precision format

Double precision numbers are stored comparable with single precision numbers.

Basic format double precision number:

seeeeeeeeeemmmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa			

The formula for double precision floating point numbers is:

$$\text{value} = (-1)^s \cdot \left(1 + \frac{m}{2^{52}}\right) \cdot 2^{e - 1023}$$

3.16.3 STORAGE IN MEMORY

Floating-point numbers are stored in IEEE754-format. Single precisions (float) and double precision (double) are stored in memory as shown below:

Address	+0	+1	+2	+3	+4	+5	+6	+7
Single	emmmmmmm	seeeeeee	mmmmmmmm	mmmmmmmm	....	....	....	....
Double	eeeemmmm	seeeeeee	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm
s = sign, e = exponent, m = mantissa, . = not used								

Single precisions numbers can be stored in a register pair. In this case the format is:

First register	Second register
seeeeeeeeemmmmmmm	mmmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa	

Double precisions numbers are never stored in registers.

3.16.4 SINGLE PRECISION USAGE

Floats can be stored in memory and in registers. The floating point library subroutines pass operands and return value through registers.

3.16.4.1 FLOAT BASE EXPRESSION SUBROUTINES

Operands, return value

The first operand is stored in R4/R5 in IEEE-754 format:

R4	R5
seeeeeeeemmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa	

The second operand is stored in R10/R11:

R10	R11
seeeeeeeemmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa	

The result is stored in R4/R5 again:

R4	R5
seeeeeeeemmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa	

Available float base expression subroutines

Subroutine	Operation	Operands	Result
__adf4r	float addition	R4R5, R10R11	R4R5
__cmf4r	float comparison	R4R5, R10R11	R4
__dvf4r	float division	R4R5, R10R11	R4R5
__mlf4r	float multiplication	R4R5, R10R11	R4R5
__sbf4r	float subtraction	R4R5, R10R11	R4R5

Table 3-16: Float base expression subroutines

3.16.4.2 FLOAT CONVERSION SUBROUTINES

Operands, return value

The single precision operand or return value is stored in R4/R5:

R4	R5
seeeeeeeemmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa	

Available float conversion subroutines

Subroutine	Operation	Operands	Result
__cff48r [1]	float to double conversion	R4R5	[R10+#*]
__cff84r	double to float conversion	[R10+#*]	R4R5
__cfi42r	float to signed int conversion	R4R5	R4
__cfi44r	float to signed long conversion	R4R5	R5R4 [2]
__cfu42r	float to unsigned int conversion	R4R5	R4
__cfu44r	float to unsigned long conversion	R4R5	R5R4 [2]
__cif24r	signed int to float conversion	R4	R4R5
__cif44r	signed long to float conversion	R5R4 [2]	R4R5
__cuf24r	unsigned int to float conversion	R4	R4R5
__cuf44r	unsigned long to float conversion	R5R4 [2]	R4R5

Table 3-17: Float conversion subroutines

[1]= Return value on the user stack
[2]=R5R4 means that the most significant word is stored in R5.

There is no negation subroutine. Its functionality can be achieved by
"BMOVN R4.15, R4.15".

3.16.4.3 REGISTER USAGE SINGLE PRECISION

The only registers destroyed by the single precision subroutines are R1-R5
and R10-R11.

3.16.5 DOUBLE PRECISION USAGE

Double precision numbers are stored in memory. The floating point library passes operands and return values on the user stack.

3.16.5.1 DOUBLE BASE EXPRESSION SUBROUTINES

Operands, return value

The first operand is stored in IEEE-754 format on the user stack and referred to by R10:

[R10+#0]	[R10+#2]	[R10+#4]	[R10+#6]
seeeeeeeeeeemmmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa			

The second operand on the user stack is referred to by R11:

[R11+#0]	[R11+#2]	[R11+#4]	[R11+#6]
seeeeeeeeeeemmmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa			

The result is stored in the user stack area referred to by R10:

[R10+#0]	[R10+#2]	[R10+#4]	[R10+#6]
seeeeeeeeeeemmmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa			

Available double base expression subroutines

Subroutine	Operation	Operands	Result
__adf8r	double addition	[R10+#*], [R11+#*]	[R10+#*]
__cmf8r	double comparison	[R10+#*], [R11+#*]	R4
__dvf8r	double division	[R10+#*], [R11+#*]	[R10+#*]
__mlf8r	double multiplication	[R10+#*], [R11+#*]	[R10+#*]
__ngf8r	double negation	[R10+#*]	[R10+#*]
__sbf8r	double addition	[R10+#*], [R11+#*]	[R10+#*]

Table 3-18: Double base expression subroutines

• • • • • • • • •

3.16.5.2   DOUBLE CONVERSION SUBROUTINES

Operands, return value

The double precision operand or return value is referred to by R10:

[R10+#0]	[R10+#2]	[R10+#4]	[R10+#6]
seeeeeeeeeeemmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm	mmmmmmmmmmmmmmmm
s = sign, e = exponent, m = mantissa			

Available double conversion subroutines

Subroutine	Operation	Operands	Result
__cff48r [1]	float to double conversion	R4R5	[R10+#*]
__cff84r	double to float conversion	[R10+#*]	R4R5
__cfi82r	double to signed int conversion	[R10+#*]	R4
__cfi84r	double to signed long conversion	[R10+#*]	R5R4 [2]
__cfu82r	double to unsigned int conversion	[R10+#*]	R4
__cfu84r	double to unsigned long conversion	[R10+#*]	R5R4 [2]
__cif28r [1]	signed int to double conversion	R4	[R10+#*]
__cif48r [1]	signed long to double conversion	R5R4 [2]	[R10+#*]
__cuf28r [1]	unsigned int to double conversion	R4	[R10+#*]
__cuf48r [1]	unsigned long to double conversion	R5R4 [2]	[R10+#*]

Table 3-19: Double conversion subroutines

[1] Return value on the user stack
[2] R5R4 means that the most significant word is stored in R5.

3.16.5.3 DOUBLE SUPPORT SUBROUTINES

Doubles can be stored anywhere in memory (near/far/huge/shuge) but the floating point library expects them to be on the user stack. This is why some library subroutines were implemented for fast copying of doubles to and from user stack.

__load8n, __load8f and __load8h copy doubles from near, far or (s)huge area to the user stack space allocated by these routines themselves. These routines change the user stack pointer and return a register pointer to the user stack.

__store8n, __store8f and __store8h copy doubles from the user stack to near, far, huge or shuge. These routines do not free the user stack space allocated by __load8*x*.

__ld0f8r and __ld1f8r allocate user stack similar to __load8*x* and copy the value 0.0 or 1.0 to this area.

Available double support subroutines

Subroutine	Operation	Operands	Result
__load8f	copy double to user stack (far)	R5R4 [1]	R10
__load8h	copy double to user stack (huge/shuge)	R5R4 [1]	R10
__load8n	copy double to user stack (near)	R4	R10
__ld0f8r	create 0.0 on alloacted user stack	None	R10
__ld1f8r	create 1.0 on allocated user stack	None	R10
__store8f	copy double from user stack to far	R10, R5R4 [1]	None, destroys R10
__store8h	copy double from user stack to huge/shuge	R10, R5R4 [1]	None, destroys R10
__store8n	copy double from user stack to near	R10, R4	None, destroys R10

Table 3-20: Double support subroutines

[1]R5R4 means that the most significant word is stored in R5.

• • • • • • • • •

3.16.5.4 REGISTER USAGE DOUBLE PRECISION

The only registers destroyed by the normal double precision subroutines are R1-R5. The input operands [R10+#*] and [R11+#*] are destroyed. R10 and R11 keep their value though, except for routines converting to double.

Usually __load8*x* and __store8*x* are also called. __load8*x* changes R0-R5 and R10, __store8*x* changes R1-R5 and R10. The subroutines __ld*x*f8r change R0-R5 and R10.

3.16.6 FLOAT/DOUBLE USAGE FOR ASSEMBLY PROGRAMMERS

Example of float usage for assembly programmers

```
; Create functionality of C expression:
;  flt1 += (float) 4 * PI;
    MOV   R4, #4             ; R4 contains int 4
    CALLA cc_UC, __cif24r  ; convert int 4 to float 4.0 (R4R5)
    MOV   R10, PI           ;
    MOV   R11, (PI+2)       ;
;                           ; R4R5: 4.0
;                           ; R10R11: PI
    CALLA cc_UC, __mlf4r    ; multiplication, result stored in R4R5
    MOV   R10, _flt1        ;
    MOV   R11, (_flt1+2)    ;
;                           ; R4R5: 4.0 * PI
;                           ; R10R11: copy of _flt1
    CALLA cc_UC, __adf4r    ; addition, result stored in R4R5
    MOV   _flt1, R4         ;
    MOV   (_flt1+2), R5      ; save result

PI: DW    04049h, 00FDBh   ; 3.141592654 (IEEE754-format)

; Registers not destroyed in this code fragment: R0, R6-R9, R12-R15
```

Example of double usage for assembly programmers

```
; Create functionality of C expression:
;  dbl1 += (double) 4 * PI;
        MOV   R4, #4          ; R4 contains int 4
allo1: CALLA cc_UC, __cif28r  ; convert int 4 to double 4.0
;                             ; ([R10+#*])
        MOV   R11, R10         ; copy pointer to 4.0 to R11
        MOV   R4, #PI           ; pointer to PI (source address)
allo2: CALLA cc_UC, __load8n  ; copy PI to new allocated stack
;                             ; ([R10+#*])
;                             ; [R10+#*]: PI (user stack)
;                             ; [R11+#*]: 4.0 (user stack)
        CALLA cc_UC, __mlf8r  ; multiplication, result stored
;                             ; in [R10+#*]
        MOV   R11, R10         ; copy pointer to 4.0 * PI to R11
        MOV   R4, #_dbl1       ;
allo3: CALLA cc_UC, __load8n  ; copy _dbl1 to new allocated stack
;                             ; ([R10+#*])
;                             ; [R10+#*]: copy of _dbl (user stack)
;                             ; [R11+#*]: 4.0 * PI (user stack)
        CALLA cc_UC, __adf8r  ; addition, result stored in [R10+#*]
        MOV   R4, #_dbl1      ; destination address in R4
        CALLA cc_UC, __store8n ; copy result to _dbl1
        ADD   R0, #24         ; restore stack
;                             ; stack allocated by lines allo*.

PI:   DW    04009h, 021FBh    ; 3.141592654 (IEEE754-format)
      DW    05452h, 04550h    ;

; Registers not destroyed in this code fragment: R0, R6-R9,
; R12-R15.
```

3.16.7 FLOATING POINT TRAPPING

Two sets of floating point libraries are delivered with the compiler, one with a floating point trapping mechanism and one without a floating point trapping mechanism (the chapter *Libraries* explains the naming conventions).

The floating point libraries with a trapping mechanism call a trapping routine which is in module trap.obj. You can replace this routine with your own trapping routine, or link your own trap routine to your application. Default, the trapping routine as delivered with the floating point libraries will never return. The infinite loop on a public label called __FPTRAPLOOP is easy to find in a debug session. See the listing of the trap handler in figure 3-6 of section 3.16.8, *Handling Floating Point Traps in a C Application*.

• • • • • • • • •

A floating point routine calls the trap routine if an error condition occurs. The type of error is specified by a trap code which is passed via register R1 to the trap routine. The result of a floating point operation is not undefined in an error situation. On error the result will be a special floating point number, such as infinite, not a number etc., except when a floating point underflow or overflow occurs.

The following table lists all the trap codes and the corresponding error description and result:

Error Description	Trap code	Result float/(unsigned) integer
Integer overflow	3	0x7FFF or 0x8000 (integer result) 0xFFFF or 0x0000 (unsigned integer result)
Floating overflow	4	+INF or –INF (float result)
Floating underflow	5	0.0 (float result)
Divide by zero	7	+INF or –INF or NaN (float result)
Undefined float	9	NaN (float result)
Conversion error	10	0 (integer result)
INF Infinite which is the largest absolute floating point number. NaN Not a Number, special notation for undefined floating point number.		

Table 3-21: Trap Codes

LANGUAGE

3.16.8 HANDLING FLOATING POINT TRAPS IN A C APPLICATION

This section explains how program execution can be continued after a floating point trap. And how floating point trap codes are passed from the floating point trap handler to a C application.

Only the floating point libraries which perform floating point trapping contain a floating point trap stub. This floating point trap stub loops infinitely, which is very helpful when you want to find a bug in your application. But when it is expected or allowed or even wanted that floating point operations generate results that are out of range, then program execution must continue after entering the floating point trap handler.

It is not possible to simply return from the floating point trap handler, because the floating point accumulator(s) contain a value which is out of range. In the same floating point operation or else in a next floating point operation there will be another call to the floating point trap handler, because the value in the floating point accumulator(s) remain out of range. This results in a succession of floating point traps.

It is impossible to assign a value to the floating point accumulator(s) which is in range and then continue program execution. If you try to assign a value to the floating point accumulators the result will always be undefined.

Interpretation of the error condition in the floating point trap handler and then continuing the floating point operation will result in most cases in a new error condition or unpredictable result. So, this is not a good solution to handle floating point error situations.

It is better to stop immediately the floating point operation which causes the floating point trap, by returning back to your application and there decide what to do with the floating point error condition. Therefore, you have to predefine an environment in your application to return to. Simply jumping back is not possible because the system-stack and user-stack are then corrupted. The floating point trap code must also be returned to the application to examine the cause of the trap.

An environment to return to in an application can be saved with the C library function `setjmp`. The C library function `longjmp` can be used in the floating point trap handler to return immediately to this saved environment. The `longjmp` restores the stack pointers, jumps back and passes the trap code to be processed.

The C listing below shows how to save an environment with `setjmp`. The assembly listing of the floating point trap handler below shows how `longjmp` is used to return to the saved environment.

There are several ways to write a C function which handles floating point traps using `setjmp` and `longjmp`. Always keep in mind that the `longjmp` function restores the environment saved by the most recent invocation of the `setjmp` function. And the environment must be saved before the `longjmp` function is called by the floating point trap handler, else program execution will be undefined.

LANGUAGE

```
/*
 * Example program which handles floating point traps by printing
 * the floating point trap code on stdout. See also floating point
 * trap handler in module trap.asm
 */
#include <stdio.h>
#include <setjmp.h>

/* Floating point environment buffer declared in trap handler */
extern jmp_buf _FP_ENV;

void
main( void )
{
   int exception;

   /*
    * Do not use floating point operations before this if
    * statement, because there is no environment saved to jump to.
    * The trap handler loops infinite when a floating
    * point operation is called from this point which traps!
    */
   /*
    * When the setjmp function has saved the environment it returns
    * zero into the exception variable, so the floating point
    * operations are executed. But if a floating point trap occurs,
    * the trap handler calls the function longjmp.
    * The longjmp function restores the environment and returns the
    * trap code in the exception variable. The trap code is a
    * non-zero value, so the else part of this if statement will be
    * executed on a floating  point trap.
    */
   if( !( exception = setjmp( _FP_ENV ) ) )
   {
      /*
       * Insert your floating point operations here.
       */
   } else
   {
      /* The exception code is a non-zero value. */
      printf("Floating point exception: %d\n", exception );
   }

   /*
    * When there is a floating point operation after this if
    * statement and it generates a floating point trap. Then the
    * program execution also continues in the else part of this if
    * statement, because the environment buffer was saved to it !
    */
}
```

Figure 3-6: Example floating point trap handling (C listing)

● ● ● ● ● ● ● ● ● ○

The floating point trap handler described by the assembly listing in figure
3-7 is archived in the floating point libraries.

```
$case
$genonly
;********************************************************************
;*
;* MODULE     : trap.asm
;*
;* APPLICATION : Floating point library 80166
;*
;* DESCRIPTION : Floating point trap handler which uses longjmp to
;*               return to a previous saved environment or loops
;*               infinite when no environment is save to return to.
;*
;* INPUT     : Register R1 contains the trap code
;*
;*      Trap code       R1,old  R1,IEEE     Description
;*      EIOVFL                  3           ; Integer overflow
;*      EFOVFL          4       4           ; Float overflow
;*      EFUNFL          5       8           ; Float underflow
;*      EFDIVZ          7       2           ; Float division by zero
;*      EFUND/EFINVOP   9       1           ; Float invalid operation
;*      ECONV           10      32          ; Conversion error
;*      ESTKUN          11           ; Floating point stack underflow
;*      ESTKOV          12           ; Floating point stack overflow
;*      EFINEXCT        16           ;
;*
;* ANALIST   : Guus Jansman
;*
;* Copyright 1991-2002 Altium BV
;*
;********************************************************************
$INCLUDE( head.asm )

@IF( @NES(@MODEL,"TINY") & @NES(@MODEL,"SMALL") )
ASSUME  DPP2:__FP_ENV      ; near data addressed via DPP2
@ENDI

PUBLIC __fptrap8            ; public declaration trapping routine
                           ; for double precision.

PUBLIC __fptrap4            ; public declaration trapping routine
                           ; for single precision.

PUBLIC __FP_ENV            ; public declaration floating point
                           ; environment buffer

PUBLIC __FPTRAPLOOP        ; public declaration trap loop

@IF( @EQS(@MODEL,"TINY") | @EQS(@MODEL,"MEDIUM") )
EXTERN  _longjmp:NEAR
@ELSE
EXTERN  _longjmp:FAR
```

```
            @ENDI


            __FPCODE  SECTION CODE WORD PUBLIC 'CPROGRAM'

            ;*****************************************************************
            ;* floating point trap handler
            ;*****************************************************************
            @IF( @EQS(@MODEL,"TINY") | @EQS(@MODEL,"MEDIUM") )
            __fptrap8 PROC NEAR
            @ELSE
            __fptrap8 PROC FAR
            @ENDI
            __fptrap4:  ; entry floating point trapping routine for single
                        ; precision operations.

               : There is no environment to return to, when the longjump return
               ; address is not set in the floating point jump buffer.
               mov   R12, (__FP_ENV)   ;if( _FP_ENV.return_address == NULL )
            @IF( @NES(@MODEL,"TINY") & @NES(@MODEL,"MEDIUM") )
               or    R12, (__FP_ENV+2) ;
            @ENDI
               jmpr   cc_Z, __FPTRAPLOOP   ; goto infinite loop
               @_JMPRACACHE

            @IF( @EQS(@MODEL,"TINY") | @EQS(@MODEL,"SMALL") )
               mov   R12, #__FP_ENV       ; R12 passes environment address
                                          ; buffer to longjmp
               mov   R13, fptrap          ; R13 passes trap code to longjmp
            @IF( @FPEXC_OP )
               mov   R14, fpexcop         ; R14 passes exception operation
            @ENDI
            @ELSE
               mov   R12, #POF (__FP_ENV) ; R12-R13 passes environment address
               mov   R13, #PAG (__FP_ENV) ; buffer to longjmp
               mov   R14, fptrap          ; R14 passes trap code to longjmp
            @IF( @FPEXC_OP )
               mov   R15, fpexcop         ; R15 passes exception operation
            @ENDI
            @ENDI

            ; restore environment loaded in the environment buffer _FP_ENV and
            ; return the trap code by calling longjmp
            @IF( @EQS(@MODEL,"TINY") | @EQS(@MODEL,"MEDIUM") )
               jmpa    cc_UC, _longjmp
            @ELSE
               @_STBUS1( _longjmp )
            @ENDI

               ; loop infinite if no environment set to return to.
            __FPTRAPLOOP:
               jmpa    CC_UC, __FPTRAPLOOP
               RETV                  ; virtual return
```

```
        __fptrap8 ENDP

        __FPCODE    ENDS


        ;********************************************************************
        ;* data section for floating point environment buffer which is
        ;* cleared at startup with C166_BSS. jmp_buf _FP_ENV;
        ;********************************************************************
        @IF( @EQS(@MODEL, "TINY") | @EQS(@MODEL, "SMALL" ) )
        __FP_ENV_BUF    SECTION   LDAT WORD PUBLIC 'CNEAR'
        @ELSE
        __FP_ENV_BUF    SECTION DATA WORD PUBLIC 'CNEAR'
        @ENDI
        __FP_ENV   LABEL   WORD
           DS   16                    ; sizeof( jmp_buf )
        __FP_ENV_BUF   ENDS

        @IF( @EQS(@MODEL, "TINY") )
        C166_BSS   SECTION   LDAT WORD GLOBAL 'CINITROM'
           DW   05h                   ; init code 05, linear data
           DW   __FP_ENV              ; start address buffer
           DW   16                    ; number of bytes to clear
        C166_BSS   ENDS
        @ENDI
        @IF( @EQS(@MODEL, "SMALL") )
        C166_BSS   SECTION PDAT WORD GLOBAL 'CINITROM'
           DW   06h                   ; init code 06, paged data
           DPPTR __FP_ENV             ; start address buffer
           DW   16                    ; number of bytes to clear
        C166_BSS   ENDS
        @ENDI
        @IF( @NES(@MODEL, "TINY") & @NES(@MODEL, "SMALL") )
        C166_DGROUP   DGROUP   __FP_ENV_BUF   ; add to default data group
        C166_BSS   SECTION DATA WORD GLOBAL 'CINITROM'
           DW   06h                   ; init code 06, paged data
           DPPTR __FP_ENV             ; start address buffer
           DW   16                    ; number of bytes to clear
        C166_BSS   ENDS
        @ENDI

        @IF( @EQS(@MODEL,"TINY") | @EQS(@MODEL,"SMALL") )
           REGDEF   R1, R12-R13
        @ELSE
           REGDEF   R1, R12-R14
        @ENDI

           END
```

Figure 3-7: Floating point trap handling (assembly-listing)

The floating point trap handler checks if an environment is set in __FP_ENV to return to. When the return address contains a NULL pointer it is supposed that there is no environment set and the trap handler continues looping infinitely. When a return address is set, the address of the jump buffer __FP_ENV and the trap code are passed to `longjmp`. Calling the `longjmp` function at the end of the trap handler restores the environment saved in __FP_ENV.

The data section containing the floating point jump buffer __FP_ENV is cleared at startup. The initialization codes for it are stored in the C166_BSS sections.

There are two entry points available in the floating point trap handler, one for double precision floating point functions causing a trap, and one for single precision floating point functions causing a trap. This default trap handler is precision independent, but if you want to write a trap handler for each precision you need these two entry points.

You can use your own floating point trap handler by linking the object module, overruling the floating point trap handler of the floating point library. Or you can replace the floating point trap object module in the floating point library with the object module of your own floating point trap handler.

3.16.9 IEEE-754 COMPLIANT ERROR HANDLING

When using the floating point libraries without trapping, the routines continue calculation with erroneous input values. This behavior is not conforming to the IEEE-754 standard, but does deliver the highest speed because the input value checking is omitted.

If your application requires IEEE-754 compliant handling of erroneous input values, the trapping version of the floating point libraries should be used. But if you do not want to handle the error conditions with a trap routine, but just continue calculation conform to IEEE-754, you can provide an empty trap function. You can add the following trap handling code to your application to achieve this:

```
#include <setjmp.h>

#pragma noclear
jmp_buf _FP_ENV;

void _fptrap8( void ) /* double precision */
{
}

void _fptrap4( void ) /* single precision */
{
}
```

LANGUAGE

3.17 INTRINSIC FUNCTIONS

When you want to use specific C166/ST10 instructions that have no equivalence in C, you normally must write (inline) assembly to perform these tasks. However, **c166** offers a way of handling this in C. The **c166** has a number of built-in functions that are implemented as intrinsic functions. The advantage of this approach is that the same C source can be compiled by a standard ANSI C compiler for simulator purposes. See section 3.21, *Portable C Code* for details.

Because the ANSI specification states that public C names starting with an underscore are implementation defined, all intrinsic functions names have a leading underscore.

Several of the intrinsic functions have restricted operand types. There are two possible restricted types. The first is called ICE which denotes that the operand must be a Integral Constant Expression rather than any type of integral expression, this is because the BMOV instruction et al do not support otherwise. The second is called BITADDR which means that the operand must be a bit addressable integer (i.e. bitword, bitaddressable sfr or bitaddressable esfr) object.

c166 has the following intrinsic functions:

_CoABS

```
void _CoABS( void );
```

Use the CoABS instruction to change the MAC accumulator's contents to its absolute value. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoABS();
CoABS
```

_CoADD

```
void _CoADD( long x );
```

Use the CoADD instruction to add a 32-bit value to the MAC accumulator. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoADD( arg1 );
CoADD R12, R13
```

_CoADD2

```
void _CoADD2( long x );
```

Use the CoADD2 instruction to add a 32-bit value, multiplied by two, to the MAC accumulator. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoADD2( arg1 );
CoADD2 R12, R13
```

_CoASHR

```
void _CoASHR( unsigned int count );
```

Use the CoASHR instruction to (arithmetic) shift right the contents of the MAC accumulator *count* times. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

The CoASHR instruction has a maximum value for *count*. Check your CPU manual for the CoASHR behaviour for large arguments.

Returns   nothing.

```
_CoASHR( 2 );
CoASHR #02h
```

LANGUAGE

_CoCMP

```
unsigned int _CoCMP( long x );
```

Inline code is generated by the C compiler to compare the MAC
accumulator contents with a 32-bit value. The returned value is a copy of
the MSW register. Only available when the MAC instruction set is enabled
with the compiler option **–xd**, **–x2** or **–x22**.

Returns   copy of MSW register.

```
isequal = _CoCMP( arg1 ) & 0x0200;
CoCMP R12, R13
CoSTORE  R4, MSW
AND      R4, #0200h
```

_CoLOAD

```
void _CoLOAD( long x );
```

Use the CoLOAD instruction to copy a 32-bit value to the MAC
accumulator. Only available when the MAC instruction set is enabled with
the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoLOAD( arg1 );
CoLOAD R12, R13
```

_CoLOAD2

```
void _CoLOAD2( long x );
```

Use the CoLOAD2 instruction to copy a 32-bit value, multiplied by two, to
the MAC accumulator. Only available when the MAC instruction set is
enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoLOAD2( arg1 );
CoLOAD2  R12, R13
```

_CoMAC

```
void _CoMAC( int x, int y );
```

Use the CoMAC instruction to add the multiplication result of two signed 16-bit values to the MAC accumulator. Only available when the MAC instruction set is enabled with the compiler option **–xd**. Note that the MP flag influences the result (it is highly recommended to keep the MP flag cleared).

Returns   nothing.

```
_CoMAC( arg1, arg2 );
CoMAC  R12, R13
```

_CoMACsu

```
void _CoMACsu( int x, unsigned int y );
```

Use the CoMACsu instruction to add the multiplication result of a signed 16-bit value with an unsigned 16-bit value to the MAC accumulator. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoMACsu( arg1, arg2 );
CoMACsu  R12, R13
```

_CoMACu

```
void _CoMACu( unsigned int x, unsigned int y );
```

Use the CoMACu instruction to add the multiplication result of two unsigned 16-bit values to the MAC accumulator. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoMACu( arg1, arg2 );
CoMACu  R12, R13
```

LANGUAGE

_CoMAC_min

```
void _CoMAC_min( int x, int y );
```

Use the CoMAC- instruction to subtract the multiplication result of two signed 16-bit values from the MAC accumulator. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**. Note that the MP flag influences the result (it is highly recommended to keep the MP flag cleared).

Returns nothing.

```
_CoMAC_min( arg1, arg2 );
CoMAC-  R12, R13
```

_CoMACsu_min

```
void _CoMACsu_min( int x, unsigned int y );
```

Use the CoMACsu- instruction to subtract the mulatiplication result of a signed 16-bit value with an unsigned 16-bit value from the MAC accumulator. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns nothing.

```
_CoMACsu_min( arg1, arg2 );
CoMACsu-  R12, R13
```

_CoMACu_min

```
void _CoMACu_min( unsigned int x, unsigned int y );
```

Use the CoMACu- instruction to subtract the multiplication result of two unsigned 16-bit values from the MAC accumulator. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns nothing.

```
_CoMACu_min( arg1, arg2 );
CoMACu-  R12, R13
```

_CoMAX

```
void _CoMAX( long x );
```

Use the CoMAX instruction to change the MAC accumulator's contents if its value is lower than the argument's value. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoMAX( arg1 );
CoMAX  R12, R13
```

_CoMIN

```
void _CoMIN( long x );
```

Use the CoMIN instruction to change the MAC accumulator's contents if its value is higher than the argument's value. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoMIN( arg1 );
CoMIN  R12, R13
```

_CoMUL

```
void _CoMUL( int x, int y );
```

Use the CoMUL instruction to store the multiplication result of two signed 16-bit values in the MAC accumulator. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**. Note that the MP flag influences the result (it is highly recommended to keep the MP flag cleared).

Returns   nothing.

```
_CoMUL( arg1, arg2 );
CoMUL  R12, R13
```

LANGUAGE

_CoMULsu

```
void _CoMULsu( int x, unsigned int y );
```

Use the CoMULsu instruction to store the multiplication result of a signed
16-bit value with an unsigned 16-bit value in the MAC accumulator. Only
available when the MAC instruction set is enabled with the compiler
option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoMULsu( arg1, arg2 );
CoMULsu  R12, R13
```

_CoMULu

```
void _CoMULu( unsigned int x, unsigned int y );
```

Use the CoMULu instruction to store the multiplication result of two
unsigned 16-bit values in the MAC accumulator. Only available when the
MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoMULu( arg1, arg2 );
CoMULu  R12, R13
```

_CoNEG

```
void _CoNEG( void );
```

Use the CoNEG instruction to change the MAC accumulator's contents to
its negated value. Only available when the MAC instruction set is enabled
with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoNEG();
CoNEG
```

_CoNOP

```
void _CoNOP( void );
```

A CoNOP instruction is generated. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoNOP();
CoNOP  [R0]
```

_CoRND

```
void _CoRND( void );
```

Use the CoRND semi-instruction to change the MAC accumulator's contents to its rounded value. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoRND();
CoRND
```

_CoSHL

```
void _CoSHL( unsigned int count );
```

Use the CoSHL instruction to shift left the contents of the MAC accumulator *count* times. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

The CoSHL instruction has a maximum value for *count*. Check your CPU manual for the CoSHL behaviour for large arguments.

Returns   nothing.

```
_CoSHL( 2 );
CoSHL  #02h
```

LANGUAGE

_CoSHR

```
void _CoSHR( unsigned int count );
```

Use the CoSHR instruction to (logical) shift right the contents of the MAC accumulator *count* times. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

The CoSHR instruction has a maximum value for *count*. Check your CPU manual for the CoSHR behaviour for large arguments.

Returns   nothing.

```
_CoSHR( 2 );
CoSHR  #02h
```

_CoSTORE

```
long _CoSTORE( void );
```

Use the CoSTORE instruction to retrieve the 32-bit value, stored in the MAC accumulator MAH and MAL. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   32-bit value from MAH and MAL.

```
x = _CoSTORE();
CoSTORE  R13, MAH
CoSTORE  R12, MAL
```

_CoSTOREMAH

```
int _CoSTOREMAH( void );
```

Use the CoSTORE instruction to retrieve the 16-bit value, stored in MAH. Only available when the MAC instruction set is enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   16-bit value from MAH

```
x = _CoSTOREMAH();
CoSTORE  R12, MAH
```

_CoSTOREMAL

```
int _CoSTOREMAL( void );
```

Use the CoSTORE instruction to retrieve the 16-bit value, stored in MAL.
Only available when the MAC instruction set is enabled with the compiler
option **–xd**, **–x2** or **–x22**.

Returns   16-bit value from MAL

```
x = _CoSTOREMAL();
CoSTORE  R12, MAL
```

_CoSTOREMAS

```
int _CoSTOREMAS( void );
```

Use the CoSTORE instruction to retrieve the 16-bit value, stored in MAS.
Only available when the MAC instruction set is enabled with the compiler
option **–xd**, **–x2** or **–x22**.

Returns   16-bit value from MAS

```
x = _CoSTOREMAS();
CoSTORE  R12, MAS
```

_CoSTOREMSW

```
int _CoSTOREMSW( void );
```

Use the CoSTORE instruction to retrieve the 16-bit value, stored in MSW.
Only available when the MAC instruction set is enabled with the compiler
option **–xd**, **–x2** or **–x22**.

Returns   16-bit value from MSW.

```
x = _CoSTOREMSW();
CoSTORE  R12, MSW
```

LANGUAGE

_CoSUB

```
void _CoSUB( long x );
```

Use the CoSUB instruction to subtract a 32-bit value from the MAC
accumulator. Only available when the MAC instruction set is enabled with
the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoSUB( arg1 );
CoSUB  R12, R13
```

_CoSUB2

```
void _CoSUB2( long x );
```

Use the CoSUB2 instruction to subtract a 32-bit value, multiplied by two,
from the MAC accumulator. Only available when the MAC instruction set is
enabled with the compiler option **–xd**, **–x2** or **–x22**.

Returns   nothing.

```
_CoSUB2( arg1 );
CoSUB2  R12, R13
```

_rol

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

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

Returns   the result.

```
      sj = _rol( ri, 4 );
MOV   R5,R9
ROL   R5,#04h
MOV   _sj,R5
```

_ror

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

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

Returns   the result.

```
      sj = _ror( si, pi );
MOV   R4,_si
ROR   R4,R12
MOV   _sj,R4
```

_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.

```
      if ( _testclear( b ) )
      BSET  USR0
      JBC   _b,_7
      BCLR  USR0
_7:
      JNB   USR0,_3
            { /* success: semaphore 'b' was free (1)
               * and now used for our critical region
               * (set to 0). Note that the code of this
               * action may be longer than 127 words
               */
            g();
      CALLA cc_UC,_g
            b = 1;      /* end critical actions: free
                         * semaphore */
      BSET  _b
            }
_3:
```

_testset

```
_bit _testset( _bit semaphore );
```

Read and set *semaphore* using the JNBS instruction.

Returns   0 if *semaphore* was not set by the JNBS instruction, 1
             otherwise.

```
            if ( _testset( b ) )
      BSET   USR0
      JNBS   _b,_8
      BCLR   USR0
  _8:
      JNB    USR0,_5
             { /* success: semaphore 'b' was free (0)
                * and now used for our critical region
                * (set to 1). Note that the code of this
                * action may be longer than 127 words
                */
             g();
      CALLA cc_UC,_g
             b = 0;       /* end critical actions: free
                            * semaphore */
      BCLR   _b
             }
  _5:
```

_bfld

```
void _bfld( BITADDR operand, ICE mask, ICE value );
```

Use the BFLDL/BFLDH instructions to assign the constant *value* to the
bit-field indicated by the constant *mask* of the bitaddressable *operand*.

```
      _bfld( bw, 0x7f, 1 );
BFLDL _bw,#07Fh,#01h
      _bfld( S0CON, 0x7f00, 0x100 );
BFLDH S0CON,#07Fh,#01h
      _bfld( bw, 0x03c0, 0x80 );
BFLDH _bw,#03h,#00h
BFLDL _bw,#0C0h,#080h
```

_getbit

```
_bit _getbit( BITADDR operand, ICE bitoffset );
```

Returns   the bit at *bitoffset* (range 0 - 15) of the bitaddressable
                    operand for usage in bit expressions.

```
        b = _getbit( P0, 0 );
BMOV    _b,P0.0
        IEN = _getbit( bwarray[2], 4 );
BMOV    IEN,_bwarray+4.4
```

_putbit

```
void _putbit( _bit value, BITADDR operand,
              ICE bitoffset );
```

Assign *value* to the bit at *bitoffset* (range 0 - 15) of the bitaddressable
operand.

```
        _putbit( 1, P0, 3 );
BSET    P0.3
        _putbit( si, P0, 2 );
MOV     R4,_si
BMOVN   P0.2,Z
        _putbit( _getbit( P0, 0 ), P0, 1 );
BMOV    P0.1,P0.0
```

_int166

```
void _int166( ICE intno );
```

Execute the C166/ST10 software interrupt specified by the interrupt
number *intno* via the software trap (TRAP) instruction. _int166(0); emits
an SRST (Software Reset) instruction.

```
        _int166( 4 );
TRAP    #04h
        _int166( 0 );
SRST
```

LANGUAGE

_idle

```
void _idle( void );
```

Use IDLE instruction to enter the idle mode. In this mode the CPU is powered down while the peripherals remain running.

Returns   nothing.

```
        if( save_power )
MOV   R5,_save_power
JMPR  cc_Z,_12
        _idle(); /* wait until peripheral interrupt
                  * or external interrupt occurs.
                  */
IDLE
_12:
```

_nop

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

A NOP instruction is generated, before and behind the nop instruction the peephole is flushed. Code generation for _nop() is exactly the same as the following inline assembly.

```
#pragma asm
    nop  ; inline nop instruction
#pragma endasm
```

Returns   nothing.

```
    value = P0;   /* read from port P0    */
MOV   R12,P0
    _nop();       /* delay for one cycle  */
NOP
    P1 = value;   /* write to port P1     */
MOV   P1,R12
```

_prior

```
unsigned int _prior( unsigned int value );
```

Use PRIOR instruction to prioritize *value*.

Returns   number of single bit shifts required to normalize *value* so
                that its MSB is set to one.

```
        register int value;
        extern int leading_zeros;

        leading_zeros = _prior( value );
    PRIOR R4,R12
    MOV   _leading_zeros,R4
```

_pwrdn

```
void _pwrdn( void );
```

Use PWRDN instruction to enter the power down mode. In this mode, all
peripherals and the CPU are powered down until an external reset occurs.

Returns   nothing.

```
            if( standby_mode )
    MOV   R4,_standby_mode
    JMPR  cc_Z,_13
            _pwrdn(); /* CPU is powered down until
                       * an external interrupt occurs.
                       */
    PWRDN
  _13:
```

_srvwdt

```
void _srvwdt( void );
```

Use SRVWDT instruction to service the watchdog timer.

Returns   nothing.

```
            _srvwdt();   /* service watchdog before
                          * it overflows.
                          */
      SRVWDT
```

_diswdt

```
    void _diswdt( void );
```

Use DISWDT instruction to disable the watchdog timer.

Returns   nothing.

```
        _diswdt();   /* disable watchdog timer */
      DISWDT
```

_einit

```
    void _einit( void );
```

Use EINIT instruction to end the initialization.

Returns   nothing.

```
        _einit();   /* end of initialization */
      EINIT
```

_atomic

```
    void _atomic( ICE number );
```

Use ATOMIC instruction to let interrupts be disabled for a specified
number of instructions (*number*=[1..4]).

Returns   nothing.

```
        _atomic( 3 );     /* next 3 instructions are
                           * not interrupted.
                           */
      ATOMIC   #03h
```

_mul32

```
long _mul32( int x, int y );
```

Use MUL instruction to perform a 16-bit by 16-bit signed multiplication and returning a signed 32-bit result. The overflow bit V is set by the CPU when the result cannot be represented in an int data type.

Returns   the result when no overflow occurs.

_mulu32

```
unsigned long _mulu32( unsigned int x,
                       unsigned int y );
```

Use MULU instruction to perform a 16-bit by 16-bit unsigned multiplication and returning a unsigned 32-bit result. The overflow bit V is set by the CPU when the result cannot be represented in an int data type.

Returns   the result when no overflow occurs.

_div32

```
int _div32( long x, int y );
```

Use DIVL instructions to perform a 32-bit by 16-bit signed division and returning a signed 16-bit result. The overflow bit V is set by the CPU when the result cannot be represented in an int data type or when the divisor **y** was zero.

Returns   the result when no overflow occurs.

_divu32

```
unsigned int _divu32( unsigned long x,
                      unsigned int y );
```

Use DIVLU instructions to perform a 32-bit by 16-bit unsigned division and returning an unsigned 16-bit result. The overflow bit V is set by the CPU when the result cannot be represented in an int data type or when the divisor **y** was zero.

Returns   the result when no overflow occurs.

LANGUAGE

_mod32

```
int _mod32( long x, int y );
```

Use DIVL instructions to perform a 32-bit by 16-bit signed modulo and returning a signed 16-bit result. The overflow bit V is set by the CPU when the quotient cannot be represented in an int data type or when the divisor **y** was zero.

Returns   the result when no overflow occurs.

_modu32

```
unsigned int _modu32( unsigned long x,
                      unsigned int y );
```

Use DIVLU instructions to perform a 32-bit by 16-bit unsigned modulo and returning a unsigned 16-bit result. The overflow bit V is set by the CPU when the quotient cannot be represented in an int data type or when the divisor **y** was zero.

Returns   the result when no overflow occurs.

```
int muldiv32( int arg1, int arg2, int divisor );

        long m32;
        int d32;

        if ( m32 = _mul32( arg1, arg2 ), V )
MOV     R8,R12
MUL     R8,R13
MOV     R9,MDH
MOV     R8,MDL
JNB     V,_14
            errno = OVERFLOW;
MOV     R4,#01h
MOV     _errno,R4
_14:

        if( d32 = _div32( m32, divisor ), V )
MOV     R15,R14
MOV     MDH,R9
MOV     MDL,R8
DIVL    R15
```

• • • • • • • • •

```
        MOV    R15,MDL
        JNB    V,_15
                    errno = OVERFLOW;
        MOV    R4,#01h
        MOV    _errno,R4
  _15:

                    return( d32 );
        MOV    R4,R15
```

_pag

```
unsigned int _pag( void * p );
```

Inline code is generated by the C compiler to get the page number of
pointer **p**. Not available in tiny model.

Returns   a 10-bit page number.

```
        pag_hp = _pag( harray );
  MOV    R4,#SOF _harray
  MOV    R5,#SEG _harray
  MOV    R12,R5
  SHL    R12,#02h
  BMOV   R12.0,R4.14
  BMOV   R12.1,R4.15
```

_pof

```
unsigned int _pof( void * p );
```

Inline code is generated by the C compiler to get the page offset of
pointer **p**. Not available in tiny model.

Returns   a 14-bit page offset.

```
        pof_hp = _pof( harray );
  MOV    R4,#SOF _harray
  MOV    R5,#SEG _harray
  MOV    R13,R4
  AND    R13,#03FFFh
```

_seg

```
unsigned int _seg( void * p );
```

Inline code is generated by the C compiler to get the segment number of pointer **p**. Not available in tiny model.

Returns   an 8-bit segment number.

```
       seg_fp = _seg( farray );
MOV    R4,#POF _farray
MOV    R5,#PAG _farray
MOV    R14,R5
SHR    R14,#02h
```

_sof

```
unsigned int _sof( void * p );
```

Inline code is generated by the C compiler to get the segment offset of pointer **p**. Not available in tiny model.

Returns   a 16-bit segment offset.

```
       sof_fp = _sof( farray );
MOV    R4,#POF _farray
MOV    R5,#PAG _farray
MOV    R15,R5
SHL    R15,#0Eh
OR     R15,R4
```

_mkfp

```
void _far *  _mkfp( unsigned int pof,
                    unsigned int pag );
```

Inline code is generated by the C compiler to make a far pointer from a page offset **pof** and page number **pag**. The arguments **pag** and **pof** are expected to be in a valid range.

Returns   a far pointer.

```
         fp = _mkfp( pof_hp, pag_hp );
 MOV    R4,R13
 MOV    R5,R12
 MOV    _fp,R4
 MOV    (_fp+2),R5
```

_mkhp

```
 void _huge *  _mkhp( unsigned int sof,
                      unsigned int seg );
```

Inline code is generated by the C compiler to make a huge pointer from a
segment offset `sof` and segment number `seg`. The arguments `sof` and
`seg` are expected to be in a valid range.

Returns   a huge pointer.

```
         hp = _mkhp( sof_fp, seg_fp );
 MOV    R5,R14
 MOV    _hp,R4
 MOV    (_hp+2),R5
```

_mksp

```
 void _shuge * _mksp( unsigned int sof,
                      unsigned int seg );
```

Inline code is generated by the C compiler to make a shuge pointer from a
segment offset `sof` and segment number `seg`. The arguments `sof` and
`seg` are expected to be in a valid range.

Returns   an shuge pointer.

Example:

The file `builtin.c` in the `c` subdirectory of the `examples` directory is a
C source file demonstrating the **c166** intrinsic functions. Compile the file
using the **–s** option to inspect generated code.

LANGUAGE

3.17.1 USER DEFINED INTRINSICS

It is possible to create user defined intrinsics. To do this you have to create a file called:

    icall.h

the compiler tries to find this file in the same way as normal include files (#include ”icall.h”) are searched. See section 4.4, *Include Files*.

In this file you can specify the prototypes of the user defined intrinsics. An intrinsic function can be defined by using the `_intrinsic` keyword, for example:

    _intrinsic float intrinsic_func(int*,long);

The `_intrinsic` keyword will only be recognized within this specific header file. It is not allowed to use preprocessor directives within this file. If this intrinsic function is called at C-level, for example:

    f=intrinsic_func(&i,l);

The compiler forces all parameters to be kept in registers, except for the parameters of type `struct/union` and `double`. Those exceptions are passed on to the user stack. Finally, the compiler generates a macro preprocessor call:

    @intrinsic_func(R8,R6,R7)

When a parameter is passed on to the user stack the stack offset of the parameter is filled in at the appropriate position, for example:

    _intrinsic void i_func(double);

will result in:

    @i_func(8)

indicating that the `double` parameter is located at stack offset 8. Parameters of the type `char` and `bit` will be passed to the macro call as 16-bit registers. Each `bit` parameter will be passed in Rx.0. An unsigned/signed `char` will be resp. zero or sign extended. The same applies to bitfield variables. The name of the macro call will always be equal to the name of the intrinsic function at C-level. The parameters will be evaluated in two groups:

1. parameters passed in registers

2. parameters passed on stack (only `doubles` and `structs/unions`)

The parameter order within these groups will not differ from the order at
C-level. The parameters passed on the user stack will be passed (and
evaluated) to the macro after the parameters that are passed in registers.
For example:

```
_intrinsic void i_func( double, struct a, int, struct b );
```

will generate the following macro call:

```
@i_func( R12, 16, 8, 0)


              ^    ^   ^  ^
              |    |   |  +-- struct b (offset  0)
              |    |   |
              |    |   +----- struct a (offset  8)
              |    |
              |    +-------- double   (offset 16)
              +----------- int
```

The macro call parameter assignments will be included in the output file
as comment, similar to the following:

```
; Macro call parameter assignments:
;
;     i1 = R12
;     l1 = R13R14
;     d1 = offset 16
;     func(ifunc(i2), d2) = offset 8
;     d2 = offset 0
;
@function(R12,R13,R14,16,8,0)
```

If a parameter occupies more than one register, all registers will be passed
separately to the macro. See the example above, where parameter 'l1' has
type 'long int'. This parameter is passed in R13/R14 at position 2 and 3
in the parameter list. If there are more registers needed then available
(max. 13) an error will be generated:

```
E 745: no registers left for expression
```

The following registers are not used for parameter passing:

- R0:   cannot be used --> User stack pointer
- R4:   cannot be used --> Used for return values/scratch
- R5:   cannot be used --> Used for return values/scratch
- USR0:   cannot be used --> Used for return values/scratch

The return value of the macro call must conform with the C166 calling convention:

Return type	Register(s)
bit	PSW.6 (USR0)
char	RL4
short/int	R4
long	R4–R5
float	R4–R5
double	(double accu on user stack)
near pointer	R4
far pointer	R4–R5
huge pointer	R4–R5
shuge pointer	R4–R5
structure	(structure on user stack)

Table 3-22: Register usage for C return types

The compiler assumes no registers to be destroyed in any case, except for the registers to pass the return value. (R4/R5/USR0 may also be used as a scratch register. You do not need to save/restore these registers).

When an intrinsic function returns a double precision floating point value or a struct/union, the compiler assumes this value at the top stack entry on return. Note that other stack space must be completely released.

The compiler will take care of copying this value to the stack location reserved for the return value, and for releasing the top stack entry. The stack space for the return value will also be reserved by the compiler before the intrinsic function is called. A typical code example is:

• • • • • • • • •

```
; test.c      30        r = double_func( f );
    SUB     R0,#08h                    ; stackspace for return value
    SUB     R0,#08h                    ; stackspace for parameter
    MOV     R12,R0
    MOV     R4,#_f
    CALLS   SEG __load8n,__load8n      ; load parameter on user stack
    MOV     R4,R12
    CALLS   SEG __store8n,__store8n    ; store parameter
    ADD     R0,#08h                    ; release space allocated by __load8n

; Macro call parameter assignments:
;
;    f = offset 0
;
    @double_func(0)                    ; intrinsic macro call
    MOV     R10,R0                     ; load source address
    MOV     R4,R0
    ADD     R4,#010h                   ; load destination address
    CALLS   SEG __store8n,__store8n    ; store return value
    ADD     R0,#08h                    ; release space for double return value
    MOV     R4,R0
    ADD     R4,#08h                    ; pointer to return value
    ADD     R0,#08h                    ; release parameter stackspace
    MOV     R10,R4
    MOV     R4,#_r                     ; destination address
    CALLS   SEG __store8n,__store8n    ; store
    ADD     R0,#08h                    ; release return value
```

For clarity, this example was compiled using **–OJ** (disabling the peephole). Normally the ADDs and SUBs on R0 are combined.

Intrinsic functions with a variable argument list are not allowed. If this occurs, the compiler generates an error:

```
E 771: variable argumentlist not allowed with
intrinsic function: "%s()"
```

There are three points that should be considered when you create an intrinsic function:

1. Special care must be taken when pointers are passed to a user defined intrinsic. When default pointers are used, the size will differ when an application is compiled in an other memory model. It is therefore advisable to specify the memory the pointer refers to and thus the pointer will always have the same size.

2. It is not possible to define pointers to intrinsic functions.

3. Internal intrinsic functions cannot be redefined.

Include a macro preprocessor file

In order to include a macro preprocessor include file you can use the following pragma:

#pragma m166include *"include-file"*

This pragma generates a $INCLUDE control in the output file. For example:

```
#pragma m166include "myinclude.inc"
```

will generate:

```
$INCLUDE(myinclude.inc)
```

On error, the following message will be generated:

```
E 744: bad #pragma m166include syntax
```

3.17.2 IMPLEMENTING OTHER _COXXX INTRINSICS USING THE _COXXX INTRINSIC FUNCTIONS

Many Co*XXX* instructions are automatically generated if a special sequence is recognized.

Examples

```
_CoLOAD( arg1 );
_CoABS();
```

generates the CoABS op1, op2 instruction.

```
_CoMUL( arg1, arg2 );
_CoRND();
```

generates the CoMUL op1, op2, rnd instruction.

```
_CoSUB( arg1 );
_CoNEG();
```

generates the CoSUBR op1, op2 instruction.

Note that the MP flag influences the result (it is highly recommended to keep the MP flag cleared).

The `CoXXXus` instructions are identical to the `CoXXXsu` variants with exchanged operands. For example, `CoMACus op1, op2, rnd` is identical to `CoMACsu op2, op1, rnd`.

The "missing" `_CoXXX` intrinsics can be defined as inline functions. For example:

```
_inline void _CoMUL_rnd( int x, int y )
    {
    _CoMUL(x,y);
    _CoRND();
    }
```

3.18 CODE MEMORY BANKING

c166 supports code memory banking. With this technique you can extend your code memory beyond 16 MB. This technique is only useful in the small and large memory model (code > 64Kb). You can specify parts (of any size) of the 16 MB of memory to use (EPROM) memory that is not addressable with a normal 24-bit address. The parts of this extra memory are called 'memory banks'.

You can use code memory banking in C by using the function qualifier:

_bank(*number*)

where, *number* is any number in the range 1 to 255.

This function qualifier uses the same syntax rules as the other function qualifiers `_interrupt(`*number*`)` and `_stackparm`. A function qualifier is allowed in both the function prototype (for the caller) and the function body itself:

```
int _bank(1) func_b1( char *, long );  /* prototype */

int _bank(2)
func_b2( int parm )   /* function body */
{
}
```

You can also use a function qualifier when you declare function pointers. The following line of C code declares a table called 'fptable' of 6 function pointers, all containing addresses of functions which are located in bank 3 and expecting their parameters (2 int types) via the user stack and returning a long:

```
long _stackparm _bank(3) (*fptable[6])( int, int );
```

Although banked interrupt functions are allowed you should not use them because they are not called as a banked function from the interrupt vector. It is recommended to make a non-banked interrupt function and call a banked function from that interrupt function.

The default situation assumes that a function is in a non-banked portion of memory (in fact _bank(0)). Valid bank numbers are 1 to 255.

• • • • • • • • •

When calling a banked function, from either non-banked memory or from a function having a different bank number, a call to a run-time library function is emitted by the C compiler instead of a regular function call. This run-time library function switches the code memory banks and calls the appropriate banked function indirectly. The code memory bank number and the inter-segment address of the banked function are passed, to the run-time library bank switch function called **__banksw**. The general purpose registers R3, R4 and R5 are used for passing these parameters. The code memory bank number of the banked function is passed in register RL3.

The current code bank number must also be passed to **__banksw**, because it might be needed to restore the code bank of the caller. Therefore the current code bank number is passed in register RH3. When RH3 is set to zero, the code bank does not need to be restored after the banked function returns. The contents of register R3 need to be saved on the user stack by the calling function, because saving it in the code bank switch function would cause a conflict with pre-calculated offsets for C function parameters and automatics. The inter-segment address of the banked function is passed in registers R4 and R5.

Code memory banking is only supported for inter-segment function calls (memory models small and large). Therefore, the **_near** keyword is not allowed with a banked function.

The following C listing displays a call to a banked function which is located in code bank 1 and called by a non-banked function. The code generated by the compiler is displayed below.

```
int _bank(1) func_b1( char *, long );

int x;
char *p;
long l;

void main( void )
{
      ...
      x = func_b1( p, l );
       ...
}
```

LANGUAGE

```
            .
            .
            .
     MOV    R12,_p              ; pass character pointer
     MOV    R13,_l              ; pass long value
     MOV    R14,(_l+2)          ;
     MOV    R4,#SOF _func_b1    ; pass inter-segment address of
     MOV    R5,#SEG _func_b1    ; banked function.
     MOV    R3,#0001H           ; pass code bank number and no
                                ; restore of current bank at
                                ; return
     MOV    [-R0],R3            ; save code bank number(s)
                                ; on the user stack
     CALLS  SEG __banksw, __banksw
                                ; call code bank switch function
     ADD    R0,#2               ; Remove code bank number(s)
                                ; from the user stack
     MOV    _x,R4               ; return result from banked
                                ; function
            .
            .
```

The default/startup situation assumes that a function is in a non-banked portion of code memory. The bank switch function and all other library functions must be located in non-banked memory, so library functions can be shared by both banked and non-banked functions.

The bank switch function may not introduce a conflict with the register usage and user stack usage implementation of C function parameter passing and C register variables. See section 3.15, *Register Usage* for details. The registers which are used for fast C parameter passing (R12-R15) may not be used by the code bank switch function and also the registers which are used for C register variables (R6-R9) may not be altered without saving them at entry and restoring them at return of the bank switch function. Register R1-R5, R10 and R11 are free for use. However, registers R4 and R5 may contain a return value from the banked function. The user stack pointer (R0) may not be changed, otherwise compiler pre-calculated offsets are affected. Keep these restrictions in mind when writing your own bank switch function. The bank switch function is a run-time library function and not a C function !

The compiler emits a special class reflecting the bank number for the code section of a banked function (e.g. class 'BANK1'). You can use these class names with the locator OVERLAY control.

• • • • • • • • •

The bank switch function depends on the hardware implementation of the code banking mechanism. There are many possible hardware implementations for code memory banking (e.g. paged, segmented etc.), this makes it impossible to write a uniform bank switch function which can be appended to the run–time library functions. Therefore a bank switch function for simulating code banking on directly accessible memory is delivered in the library. This allows to test your application on an evaluation board without having the real hardware implementation available. Finally you can use the skeleton of the delivered assembly bank switch function to write your own bank switch function, supporting your hardware implementation.

The delivered simulation routine assumes the following situation: The different code banks are located in physical memory but they are treated as if they are located in virtual c.q. banked memory. The code banking is simulated by copying the page the banked code is located in to a reserved page where the code is executed from. In fact the code bank number is treated as a page number. So, a code bank is limited to the size of one page (16Kb). One page is reserved for execution of banked code. This page cannot be used for other code or data, because it contains the currently active code page. All the code banks are overlaid in this physical code page with the locator OVERLAY control.

The following listing shows the assembly code for simulating code banking. The number of code banks is restricted to the number of pages which are available for code banking. The physical page the code banks are overlaid in and executed from is defined by the equate CODE_PAGE. The default value of CODE_PAGE is page 15. The following locator control can be used:

```
OVERLAY ( 'BANK4', 'BANK5' ( RANGEP(15) ) )
```

This control instructs the locator to overlay the classes BANK4 and BANK5 in page 15. Remember that, when using our simulation code, the code from bank 4 must be located in page 4 and the code from bank 5 must be located in page 5. You can use the regular CLASSES control to achieve this. See the description of the OVERLAY locator control in the assembler manual for a detailed example.

The actual bank switch is performed by __pgbk. In the simulation approach, the code bank number (passed via RL3) corresponds to the page number where the banked function is present. This page must be activated, which means copied to the physical page defined by CODE_PAGE. Now you can actually call the banked function, indirectly, using the run-time library function __icall. The inter-segment address of the banked function is passed in registers R4 and R5 to __icall.

The code bank number of the currently active code bank is pushed on the user stack and afterwards removed from it by the function calling the bank switch function. It is not possible to save the current code bank number on the user stack at function entry of the bank switch function, because this affects the user stack pointer, introducing a conflict with precalculated offsets for C function parameters and automatics. When code execution returns from the banked function, this code bank number is read from the user stack and, when needed, the previous code bank is reactivated by calling __pgbk again. This allows you to call a banked function from a banked function in a different code bank.

You can use the skeleton bankswh.asm, in the bank subdirectory of the examples directory, as a starting point to implement your hardware implementation of bank switching. In this case, you only have to replace the code from __pgbk with your own code, actually performing the hardware bank switch. It is obvious that your hardware bank switch approach is not limited to the size of a page.

Restriction: When a banked function (e.g. f1) calls a non-banked function (e.g. f2) which on its turn calls a banked function in another bank (e.g. f3), the original bank is not restored when returning from the non-banked function (f2).

3.19 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
```

LANGUAGE

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.

Quality Assurance report

To ensure compliance to the MISRA C rules throughout the entire project, the TASKING C166/ST10 Linker/Locator can generate a MISRA C Quality Assurance report. This report lists the various modules in the project with the respective MISRA C settings at the time of compilation. You can use this in your company's quality assurance system to provide proof that company rules for best practice programming have been applied in the particular project.

If the MISRA C error level is set to 'warnings', then the MISRA C rules are marked as checked.

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:

```
-misrac n,n,...
```

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

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

3.20 PEC SUPPORT

c166 supports the initialization of the PEC source and destination pointers using a (int) cast in C. The following example shows how to allocate a PEC-addressable section for a buffer in the first 64K segment:

```
#include <reg166.h>

#if _MODEL == 'l' || _MODEL == 'm'
#pragma align fb=c        /* declare PECADDRESSABLE data section for
                             'far' data */
#pragma class fb=firstsegment /* assign a special class name to
                                 this section                 */
char    _far buffer[100];     /* explicitly '_far', otherwise
                                 allocated in default data group */
#pragma default_attributes    /* restore default section
                                 attributes for '_far' data    */
#else
char    buffer[100];
#endif

void
f()
{
      DSTP0 = (int)buffer;    /* when you use the c++ compiler,
                                 use a long cast instead of an
                                 integer: DSTP0 = (long)buffer;  */
}
```

If large model (**–Ml**) is used, the following code is generated:

```
PEC1_1_FB  SECTION  DATA PECADDRESSABLE  PUBLIC 'firstsegment'
PEC1_1_FB_ENTRY  LABEL  BYTE
_buffer  LABEL  BYTE
      DS  100
      PUBLIC  _buffer
PEC1_1_FB  ENDS

      PUBLIC  _f
PEC1_2_PR  SECTION    CODE WORD PUBLIC 'CPROGRAM'
_f    PROC  FAR
      MOV   R4,#SOF (_buffer)
      MOV   DSTP0,R4
      RETS
_f    ENDP
PEC1_2_PR  ENDS
```

The following example shows how to allocate a PEC-addressable section for a buffer in the SYSTEM page (page 3, 16K). The SYSTEM page is in the PEC-addressable range (segment 0). Therefore, it is not needed to declare the buffer data section PECADDRASSABLE with `#pragma align sb=c`.

```
#include <reg166.h>

char _system buffer[100];      /* explicitly '_system',
                                  allocated in system page */
f()
{
    DSTP0 = (int)buffer;
}
```

If large model (**–Ml**) is used, the following code is generated:

```
    ASSUME      DPP3:SYSTEM

PEC1_1_SB  SECTION      DATA WORD PUBLIC 'CSYSTEM'
PEC1_1_SB_ENTRY  LABEL  BYTE
_buffer          LABEL  BYTE
    DS          100
    PUBLIC      _buffer
PEC1_1_SB  ENDS


    PUBLIC      _f
PEC1_2_PR  SECTION      CODE WORD PUBLIC 'CPROGRAM'
_f    PROC  FAR
    MOV   R4,#SOF _buffer
    MOV   DSTP0,R4
    RETS
_f    ENDP
PEC1_2_PR  ENDS

C166_SGROUP  DGROUP  PEC1_1_SB,SYSTEM
```

The XC16x/Super10 architecture supports a PECSEG*x* register for each PEC channel. The upper eight bits of this register are used as the segement number for SCRP*x*. The lower eight bits are used as the segment number for DSTP*x*. This allows PEC transfers between any kind of memory or register, not necessarily in segment zero. So, if you want to use any kind of segment, you should not use the PECADDRESSABLE sections. The following example shows how to initialize a Super10 PEC buffer.

```
#include <regsuper10bo.h>

_shuge int buffer[1000];

f()
{
    PECSEG0 &= 0xFF00;
    PECSEG0 |= seg( buffer );
    DSTP0 = sof( buffer );
}
```

3.21 PORTABLE C CODE

If you are developing C code for the C166/ST10 using **c166**, you might want to test the code on the host you are working on, using a C compiler for that host. Therefore, the include file `c166.h` is delivered with the compiler, which must be included in your C programs.

This header file checks if the predefined macro _C166 is defined (**c166** only). If not, all C166/ST10 language extensions (read keywords) are redefined to ANSI C equivalents. Furthermore an adapted prototype of each C166/ST10 intrinsic 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 C166/ST10 processor and link these functions with your application for simulation purposes.

If you want to isolate all functions using **c166** language extensions in separate modules, you can use the **–A** option (disable language extensions) to check if **c166** keywords are still present.

You can enable/disable groups of language extensions separately. See the description of the **–A** option in the next chapter for more information.

3.22 HOW TO PROGRAM SMART WITH C166

If you want to get the best code out of **c166**, the following guidelines should be kept in mind:

1. Always include the appropriate header file before using a standard C library function. This is very important with variable argument list functions, such as `printf()`!
   Note that you do not have to edit all the 'old style' function bodies of your application into 'new style' ANSI function bodies. You only have to add a full prototype declaration before any function is called and before any function definition.
   The following example shows how to migrate from old style programs to new style without editing the function bodies of the program. The advantage of this method is, that if 'prototyping' is not possible (because the C program must be translated with a non-ANSI compiler), the program does not have to be changed:

```
#ifdef prototyping
#define  FD(x) x         /* full function prototype */
#else
#define  FD(x) ()        /* return type only: no arguments */
#endif

char* cg_var FD( (char *, int) );
void  main FD( (void) );

void
main()
{
   char *p;

   p = cg_var( "text", 2 );
}

char *
cg_var( name, offset )
char *name;
int offset;
{
   return ( name + offset );
}
```

If 'prototyping' is enabled the function call to **cg_var** is using the full prototype and the function body of **cg_var** is treated like a 'new style' function, using the full prototype.

2. Try to use the 'unsigned' type modifier as much as possible, because it takes less code to convert an unsigned variable to a long variable than a signed variable.

3. Do NOT use the **–A** option. This option is implemented as strict ANSI conformance checking, disabling language extensions and character arithmetic code generation. This option may decrease code density and execution speed.

4. In most of the cases it is safe to use the **–Oa** option, which results in better code density. However, you have to check your application on 'aliases'. If this option is not used (default), **c166** 'forgets' all register contents bound to C variables if an indirect write operation (e.g. MOV [R4],R5) is performed.

See section 3.2.1.7, *Efficiency in Large Data Models (Medium/Large/Huge)*.

5. Use the **–Om** option (default) and non-protected library if multiply and divide instructions do not have to be protected against interrupts. This results in better code density and faster execution.

6. Use the intrinsic functions, if special C166/ST10 instructions are needed.

7. If you want to overrule the **c166** register allocation of C variables, you must use the register storage class specifier in the declaration of this (local) variable, because **c166** might allocate other C variables into the CPU registers, than the variables you prefer to be in registers.

8. Avoid static initialized bit variables (which must have the value '1' after startup), because this takes a lot of ROM space and is very time consuming during system startup.

9. Use the **–t** option, to inspect the size of the code generated. This is useful, when 'experimenting' with compiler options.

10. Use the **–Of** optimization option to prefer speed instead of code density (**–OF** is default).

11. Use the **–Ox** optimization option to enable extra inlining of C library functions when you prefer speed instead of code density.

CHAPTER 4

COMPILER USE

TASKING

4.1   CONTROL PROGRAM

The control program **cc166** is provided to facilitate the invocation of the various components of the C166/ST10 toolchain. The control program accepts source files, options and controls on the command line in random order.

The invocation syntax of the control program is:

cc166 [[*option*]... [*control*]... [*file*]...]...

Options are preceded by a '-' (minus sign). Controls are reserved words. The input *file* can have any extension as explained below.

When you use a **UNIX** shell (Bourne shell, C-shell), arguments containing special characters (such as '( )' and '?') must be enclosed with **" "** or escaped. The **-?** option (in the C-shell) becomes: **"-?"** or **-\?**.

The control program recognizes the following argument types:

- Arguments starting with a '-' character are options. Some options are interpreted by **cc166** itself; the remaining options are passed to those programs in the toolchain that accept the option.
- Arguments which are known by **cc166** as a control are passed to those programs in the toolchain that accept the control.
- Arguments with a **.cc**, **.cxx** or **.cpp** suffix are interpreted as C++ source programs and are passed to the C++ compiler.
- Arguments with a **.c** suffix are interpreted as C source programs and are passed to the compiler.
- Arguments with a **.asm** suffix are interpreted as assembly source files which are preprocessed and passed to the assembler.
- Arguments with a **.src** suffix are interpreted as preprocessed assembly source files. They are directly passed to the assembler.
- Arguments with a **.lib** suffix are interpreted as library file and passed to the link stage of **l166** when the **-cf** option is not specified. When the **-cf** is specified, the libraries are passed to the locate stage.
- Arguments with a **.ili** suffix are interpreted as linker invocation files and are passed to the link stage of **l166** with a leading '@' sign.
- Arguments with a **.ilo** suffix are interpreted as locator invocation files and are passed to the locate stage of **l166** with a leading '@' sign.

- Arguments with a `.out` suffix are interpretes as input files for the Motorola S formatter, IEEE formatter or Intel Hex formatter. Specify the formatter respectively with the options **–srec**, **–ieee** or **–ihex**.
- Everything else is considered an object file and is passed to the linker.

Normally, **cc166** tries to compile and assemble all files specified, and link and locate them into one output file. There are however, options to suppress the assembler, linker or locator stage. The control program produces unique filenames for intermediate steps in the compilation process. These files are removed afterwards. If the compiler and assembler are called in one phase, the control program prevents preprocessing of the generated assembly file. Normally assembly input files are preprocessed first.

The following options are interpreted by the control program **cc166**:

Option	Description
–?	Display invocation syntax
–V	Display version header and stop
–Waarg	Pass argument directly to the assembler
–Wcarg	Pass argument directly to the compiler
–Wcparg	Pass argument directly to the C++ compiler
–Wfarg	Pass argument directly to the object formatter
–Wlarg	Pass argument directly to the linker
–Wmarg	Pass argument directly to the macro preprocessor
–Woarg	Pass argument directly to the locator
–Wplarg	Pass argument directly to the C++ pre–linker
–c++	Force `.c` files to C++ mode
–c	Do not link: stop at `.obj`
–cc	Compile C++ files to `.c` and stop
–cf	Skip the linking phase; call the locator directly
–cl	Do not locate: stop at `.lno`
–cm	Always also invokes the C++ muncher
–cp	Always also invokes the C++ pre–linker
–cprep	Use C preprocessor instead of macro preprocessor
–cs	Do not assemble: stop at `.src`

Option	Description
–f *file*	Read arguments from *file* (”-” denotes standard input)
–gs	Pass **–cl** to **ieee166**, set compatibility mode to 1
–ieee	Produce an IEEE–695 output file
–ihex	Produce an Intel hex output file
–lib *directory*	Specify the location of user–built libraries
–libcan	Link CAN library
–libfmtio*variant*	Link MEDIUM or LARGE printf( )/scan( ) library variants
–libmac	Link MAC optimized runtime library
–noc++	Force C++ files to C mode
–nolib	Do not link with the standard libraries
–nostl	Do not link the STLport library
–nostlo	Do not link the STLport extension library
–o *file*	Specify the output file
–srec	Produce an S–record output file
–tmp	Keep intermediate files
–trap	Use a floating point library with trap handler.
–notrap	Use a floating point library without trap handler.
–v	Verbose option: show commands invoked
–v0	Same as **–v**, but commands are not started
–wc++	Enable C and assembler warnings for C++ files

Table 4-1: Control program options

For more detailed information about the control program **cc166**, refer to section *cc166* in Chapter *Utilities* of the *Cross-Assembler Linker/Locator, Utilities User's Manual*.

4.2   COMPILER

The invocation syntax of the C166 compiler is:

c166 [[*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 **a166**).

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
–B[*flag*...]	Control CPU problem bypasses
–Dmacro[=*def*]	Define preprocessor *macro*
–E[**m**|**c**|**i**|**p**|**x**]	Preprocess only
–F[*flag*...]	Control floating point
–Ggroupname	Use *groupname* to group near data sections (**–Mm**, **–Ml** or **–Mh** only)
–Hfile	Include *file* before starting compilation
–Idirectory	Look in *directory* for include files
–M{**t**|**s**|**m**|**l**|**h**}	Select memory model: tiny, small, medium, large or huge
–Oflag...	Control optimization
–P[**d**]	Use user stack model stack frame (calling convention) (to be used with special stack frame C library if '**d**' is not specified)
–R{**cl**|**co**|**al**}mem=*new*	Change **cl**ass name, **co**mbine type or **al**ign type of section for *mem*
–S	Static allocation of automatics
–Tsize	Use *size* as threshold before allocating data in default data group (**–Mm**/**–Ml**/**–Mh** only)

USAGE

Option	Description
–T[*size*],*size2*	In addition to the previous option, you can also specify a threshold for intiialized data. Default:infinite
–U*macro*	Remove preprocessor *macro*
–V	Display version header only
–e	Remove output file if compiler errors occur
–err	Send diagnostics to error list file (`.err`)
–exit	Alternative exit values
–f *file*	Read options from *file*
–g[**b**|**f**|**l**|**s**]	Enable symbolic debug information
–gso	Enable GSO (acquire phase)
–gso=*file.gso*	Enable GSO (allocation phase)
–i*scale*	Specify scaling of interrupt vector table (needs **–x2**): 　　0 – for no scaling (default) 　　1 – for x2 　　2 – for x4 　　3 – for x8
–m*mem=size*	Specify memory *size*
–m*mem=*[*size*],*n*	Specify maximum section *size* for *mem* and in addition a threshold *n* for switching to a new section
–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
–o *file*	Specify name of output *file*
–r[*name*[,**c**][,*regdef*]	Omit REGDEF or specify number (*nr*) of GPR registers, the *name* of the register bank and **c** for common
–s[**i**]	Merge C–source code with assembly output
–t	Display module summary and write section information in output file
–u	Treat all 'char' variables as unsigned
–w[*number*]	Suppress one or all warning messages
–wstrict	Suppress warning messages 183,196 and 216

Option	Description
–x[**1**|**2**|**22**|**d**]	Allow all or some functions of the extended architectures (to be used with ext or ext2 library sets)
–z*pragma*	Identical to '#pragma *pragma*' in the C source

Table 4-2: Compiler options (alphabetical)

Description	Options
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	**–E**[**m**|**c**|**i**|**p**|**x**]
Define preprocessor *macro*	**–D***macro*[=*def*]
Remove preprocessor *macro*	**–U***macro*
Allocation control options	
Use *groupname* to group near data sections (**–Mm**, **–Ml** or **–Mh** only)	**–G***groupname*
Change **cl**ass name, **co**mbine type or **al**ign type of section for *mem*	**–R**{**cl**|**co**|**al**}*mem*=*new*
Static allocation of automatics	**–S**
Use *size* as threshold before allocating data in default data group (**–Mm**/**–Ml**/**–Mh** only)	**–T***size*
In addition to the previous option, you can also specify a threshold for intiialized data. Default:infinite	**–T**[*size*]**,***size2*
Specify memory *size*	**–m***mem*=*size*
Specify maximum section *size* for *mem* and in addition a threshold *n* for switching to a new section.	**–m***mem*=[*size*]**,***n*
Code generation options	
Control CPU problem bypasses	**–B**[*flag*...]
Control floating point	**–F**[*flag*...]
Select memory model: tiny, small, medium, large or huge	**–M**{**t**|**s**|**m**|**l**|**h**}

Description	Options
Control optimization	**–O**_flag_...
Use user stack model stack frame (calling convention) (to be used with special stack frame C library if '**d**' is not specified)	**–P**[**d**]
Enable GSO (acquire phase)	**–gso**
Enable GSO (allocation phase)	**–gso**=_file.gso_

Specify scaling of interrupt vector table (needs **–x2**):
    0 – for no scaling (default)
    1 – for x2
    2 – for x4
    3 – for x8
	–i_scale_

Omit REGDEF or specify number (_nr_) of GPR registers, the _name_ of the register bank and **C** for common	**–r**[_name_[,**c**][,_regdef_]
Allow all or some functions of the extended architectures (to be used with ext or ext2 library sets)	**–x**[**1**|**2**|**22**|**d**]
Identical to '#pragma _pragma_' in the C source	**–z**_pragma_
Language control options	
Enable/disable specific language extensions	**–A**[_flag_...]
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**[**i**]
Diagnostic options	
Display invocation syntax	**–?**
Display version header only	**–V**
Send diagnostics to error list file (`.err`)	**–err**
Alternative exit values	**–exit**
Enable symbolic debug information	**–g**[**b**|**f**|**l**|**s**]
Enable individual MISRA C checks	**–misrac**_n,n,..._
Generate warnings for advisory MISRA C rules	`-misrac-advisory-warnings`

Description	Options
Generate warnings for required MISRA C rules	**–misrac-required-warnings**
Display module summary and write section information in output file	**–t**
Suppress one or all warning messages	**–w**[*number*]
Suppress warning messages 183, 196 and 216	**–wstrict**

Table 4-3: Compiler options (functional)

4.3   DETAILED DESCRIPTION OF THE COMPILER 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).

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:

   c166 –?

-A

Option:

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

–A[*flags*]

Arguments:

Optionally one or more language extension flags.

Default:

–A1

Description:

Control language extensions. Without the **–A** option all **c166** language
extensions are enabled. **–A** without any flags, specifies strict ANSI mode;
all language extensions are disabled. This is equivalent with
–ACDFIKLMPSTUVWX 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   <u>Default.</u> Perform character arithmetic. **c166** 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
     3.4.2 *Character Arithmetic*.

C   Disable character arithmetic.

d   <u>Default.</u> Define storage for uninitialized constant rom data, instead of
     implicit zero initialization. The compiler generates a 'DS 1' for 'const
     char i[1];'.

D   Uninitialized constant rom data is implicitly zero. The compiler
     generates a 'DB 1' for 'const char i[1];'.

USAGE

f Default. 14-bit arithmetic is used for far pointer comparison instead of long 32-bit arithmetic. Only the page offset is compared. Far pointers do not cross page boundaries and if the objects pointing to are not members of the same aggregate or (union) object, the result is undefined. When far pointers are compared to NULL, 32-bit arithmetic is needed !

F 32-bit arithmetic is used for far pointer comparison.

i Default. Inlining of a selected group C-library functions is allowed. This option works together with the extra inlining optimization option **–Ox**. Note: It is not possible to take the address of an inline function, which is not conform to the ANSI-C standard.

I Disable inlining of C-library functions, to conform to strict ANSI-C mode.

k Default. The keywords `_atbit`, `bank`, `bit`, `bitword`, `esfr`, `esfrbit`, `far`, `huge`, `interrupt`, `iram`, `near`, `sfr`, `sfrbit`, `stackparm`, `system` and `using` are recognized as C language extensions. See chapter 3 *Language Implementation* for the explanation of these language extensions.

K Disable all keywords which are an extension of the C language.

l Default. 500 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.

m Default. When a 32 bit value is divided by a 16 bits divisor and only 16 bits of the result are being used, then the operation can be done by a DIVL or DIVLU instruction, depending on the signed/unsigned setting of the operands. The same applies for the modulo operator. When there are chances for overflow and the (truncated) result must still be conform ANSI, then it is better to switch this option off. Example:

```
long  m32
short m16, divisor;

m16 = m32 / divisor;
m32 = (short)(m32 / m16);
```

See also the intrinsic functions _div32, _divu32, _mod32 and _modu32 in section 3.17.

M Perform divide/modulo operation always in 32 bits using run-time library calls.

p <u>Default.</u> 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 <u>Default.</u> __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 <u>Default.</u> 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 <u>Default.</u> Use type unsigned char for 0x80-0xff. The type of an octal or hexadecimal constant, not suffixed with 'L' or 'l', 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 octal or hexadecimal constant, not suffixed with 'L' or 'l', 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.

w Default. Allow propagation of **const** initializers. This optimization makes the following code possible:

```
const   int     one = 1;
int     array [] = { one };
```

W Disable propagation of **const** initializers.

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 **–ACDFIKLMPSTUVWX** (disable all).

1 Same as **–AcdfiklmpstuVwx** (default).

Example:

To disable character arithmetic and C++ comments enter:

```
c166  –ACP test.c
```

• • • • • • • • •

-B

Option:

From the **Project** menu, select **Project Options...**
Expand the **Application** entry, expand the **Processor** entry and select
CPU Problem Bypasses and Checks. Enable or disable one or more
bypasses.

–B[*flags*]

Arguments:

Optionally one or more CPU functional problem bypass flags.

Default:

–Babdefhijklmnou

Description:

Enable/disable bypass for certain CPU functional problems. Without the **–B**
option the default is **–Babdefhijklmnou** (all bypasses off).

Flags which are controlled by a letter, can be switched on with the
uppercase letter and switched off with the lowercase letter. The following
flags are allowed:

a <u>Default.</u> Do not protect DIVx/MD[LH] sequences by an ATOMIC
instruction.

A Protect DIVx/MD[LH] sequences by an ATOMIC instruction. The DIVx
instruction and a read from MDL/MDH are not interruptable because
the will be generated within the same atomic sequence. This is a
bypass for the LONDON1751 CPU functional problem. Refer to
Appendix C, *CPU Functional Problems* for details.

b <u>Default.</u> Do not place two NOP instructions after each instruction
which does a byte write. This option is equivalent to the pragma
nofix_byte_write.

B Place two NOP instructions after each instruction which does a byte
write. These instructions are: ADDB, ADDCB, ANDB, CPLB, MOVB,
NEGB, ORB, SUBB, SUBCB, XORB. This is a bypass for CPU problem
S1, as described in Appendix C, *CPU Functional Problems*. This option
is equivalent to the pragma **fix_byte_write**.

USAGE

d <u>Default</u>. Assume hardware environment is present, where there is no need to protect the execution of divide instructions against interrupts. Emit inline code (DIV) instead of a run-time library call.

D This option emits code to protect signed divide operations against interrupts. The protection will be generated inline using ATOMIC instructions. This is a bypass for the CPU problem 13, as described in Appendix C, *CPU Functional Problems*. Use the protected version of the library (`lib\[u]extp\*.lib` or `lib\[u]ext2p\*.lib`).

e <u>Default</u>. Never extend EXTEND sequence with one instruction.

E EXTEND sequences are extended with one instruction when addressing mode `Rn,[Rm + #data16]` is the last instruction of the EXTEND sequence.

   This is a bypass for the CPU.3 problem, as described in Appendix C, *CPU Functional Problems*.

f <u>Default.</u> Do not prevent the generation of `MOVB [Rn],mem` instructions.

F Disable the generation of `MOVB [Rn],mem` instructions when even 'const' objects are accessed. This is a bypass for the CPU.16 problem as described in Appendix C, *CPU Functional Problems*.

h <u>Default.</u> Do not prevent the generation of `Label_C: JMPR cc.xx, Label_A` instructions.

H Disable the generation of `Label_C: JMPR cc.xx, .Label_A` instructions. This is a bypass for the BUS.18 problem as described in Appendix C, *CPU Functional Problems*.

i <u>Default.</u> Do not place `BFLDH PSW,#0F0h,#0F0h` before `RETI` in interrupt functions

I Place the instruction `BFLDH PSW,#0F0h,#0F0h` before `RETI` in interrupt functions.

   This is a bypass for the CPU problem 17 as described in Appendix C, *CPU Functional Problems*.

j <u>Default.</u> Do not place ATOMIC #2 before a JMPS instruction. Do not delete the return addresses from the system stack in interrupt functions.

J   Place ATOMIC #2 before a JMPS instruction. The JMPS instructions in the interrupt vector table will be replaced by CALLS instructions (linker / locator control: FIXSTBUS1). The compiler generates an ADD SP, #04 instruction to delete the return address (generated by CALLS) from the system stack. This is a bypass for the ST_BUS.1 problem as described in Appendix C, *CPU Functional Problems*.

The instruction to delete the return address from the system stack is part of the interrupt frame. If **#pragma noframe** was used, this instruction will not be generated, you have to do it manually.

k   <u>Default.</u> Do not protect BFLDH/BFLDL instructions by an ATOMIC instruction.

K   Protect BFLDH/BFLDL instructions by an ATOMIC instruction. This is a bypass for the CPU.21 CPU functional problem. Refer to Appendix C, *CPU Functional Problems* for details.

l   <u>Default.</u> Do not protect JMPI/CALLI instructions by an ATOMIC instruction.

L   Protect JMPI/CALLI instructions by an ATOMIC instruction. This is a bypass for the LONDON1 CPU functional problem. Refer to Appendix C, *CPU Functional Problems* for details.

m   <u>Default</u>. Assume hardware environment is present, where there is no need to protect the execution of multiply instructions and divide instructions against interrupts. Emit inline code (MUL, DIV, DIVU, DIVL, DIVLU) instead of a run-time library call. You must use the non-protected version of the library.

M   This option emits code to protect multiply/divide operations against interrupts. The protection will be generated inline using ATOMIC instructions. Use the protected version of the library (`lib\extp\*.lib`).

This is a bypass for many CPU problems, among which are problem 7, problem 13, problem 17, CPU.2, CPU.11 and CPU.18. as described in Appendix C, *CPU Functional Problems*.

n   <u>Default</u>. Do not avoid pipeline conflict after CoSTORE instruction.

N   Avoid pipeline conflict after CoSTORE instruction. This is a bypass for the Kfm_BR03 CPU functrional problem as described in Appendix C, *CPU Functional Problems*.

o   <u>Default.</u> Do not prevent the generation of MOV(B) Rn, [Rm+#data16] instructions.

O   Disable generation of MOV(B) Rn, [Rm+#data16] instructions. The generation of this instruction is not disabled in some of the intrinsic functions since the source operand always refers to internal RAM here.

   This a bypass for the CPU1R006 functional problem, as described in Appendix C, *CPU Functional Problems*.

u   <u>Default</u>. Assume hardware environment is present, where there is no need to protect the execution of multiply instructions against interrupts. Emit inline code (MUL/MULU) instead of a run-time library call. You must use the non-protected version of the libraries (`lib\ext\*.lib`).

U   This option emits code to protect multiply operations against interrupts. The protection will be generated inline using ATOMIC instructions. Use the protected version of the libraries (`lib\extp\*.lib`).

   This is a bypass for CPU problems CPU.11 and problem 17.

Zc166sv1div
   Do not generate unprotected division instructions. This is a bypass for the CR105893 functional problem.

Zno_c166sv1div
   <u>Default.</u> Allow generation of unprotected division instructions.

Zc166sv1ext
   Do not jump from extend sequences. This is a bypass for the CR107092 functional problem.

Zno_c166sv1ext
   <u>Default.</u> Jump from extend sequences.

Zc166sv1jbc
   Do not use JBC and JNBS instructions, unless the first operand is a GPR. This is a bypass for the CR105981 functional problem.

Zno_Zc166sv1jbc
   <u>Default.</u> Always use JBC and JNBS instructions.

Zc166sv1trap
   Insert a NOP before a TRAP instruction. This is a bypass for the CR105619 functional problem.

Zno_c166sv1trap
    <u>Default.</u> Do not insert a NOP before a TRAP instruction.

Zcpu_jmpra_cache
    Fix broken program flow after not taken JMPR/JMPA instruction. This is a bypass for the CR108400 functional problem.

Zno_cpu_jmpra_cache
    <u>Default</u>. Do not fix broken program flow after not taken JMPR/JMPA instruction.

Zcpu_reti_int
    Fix lost interrupt while executing RETI instruction. This is a bypass for the CR108342 functional problem.

Zno_cpu_reti_int
    <u>Default</u>. Do not lost interrupt while executing RETI instruction.

Zinsert_div_mdl
    Insert NOP instructions between DIV and the read of MDL. This is a bypass for the CR108309 functional problem.

Zno_insert_div_mdl
    <u>Default.</u> Do not insert NOP instructions between DIV and the read of MDL.

Zinsert_mdlh_muldiv
    Insert NOP instruction between write to MDL/MDH and DIVx/MULx instruction. This is a bypass for the CR108904 functional problem.

Zno_insert_mdlh_muldiv
    <u>Default.</u> Do not insert NOP instruction between write to MDL/MDH and DIVx/MULx instruction.

Zmovbjb
    Insert NOP instruction between an explicit byte-register modification and a bit-jump that uses the opposite byte of the same word GPR.

Zno_movbjb
    <u>Default.</u> Do not insert NOP instruction between explicit byte-register modification and a bit-jump.

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

USAGE

-D

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Preprocessing**.
In the **Define user macro** box, click on an empty **Macro** field and enter
a macro name. Optionally, click in the **Definition** field and enter a
definition.

–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, you can use the **–f** option.

Example:

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

```
c166 –DNORAM –DPI=3.1416 test.c
```

–U

-E

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Preprocessing**.
Enable the **Store preprocessor output (<file>.i)** check box.

–E[**m**|**c**|**i**|**p**|**x**]

Description:

Run the preprocessor of the compiler 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.

An overview of the flags is given below.

m - generate dependencies for make
c - do not strip comments
i - keep #include directives
p - do not generate #line source position info
x - disable macro expansion

The **m** flag overrules all other flags.

Examples:

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

```
c166 –E –o preout test.c
```

The following command generates dependency rules for the file `test.c` which can be used by **mk166** (the C166/ST10 'make' utility).

```
c166 –Em test.c

test.obj :  test.c
```

USAGE

–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:

```
c166 –e test.c
```

-err

Option:

In EDE this option is not useful.

–err

Description:

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

Example:

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

```
c166 -err test.c
```

USAGE

-exit

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Diagnostics**.
Enable the **Exit with error status even if only warnings were generated** check box.

–exit

Description:

Use alternative exit values in case warnings are reported. In case warnings are reported, the compiler returns an exit value as if there were errors reported.

-F

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Floating Point**.
Enable or disable floating point options.

–F[*flags*]

Arguments:

Optionally a floating point control flag.

Default:

–Fs

Description:

Control floating point. The flags which are controlled by a letter can be switched on with the lowercase letter and switched off with the uppercase letter. **–F** used without flags is the same as using **–Fs**. Currently the following flags are implemented.

c   Enables the use of `float` constants.

C   <u>Default</u> This flag is ignored when **–Fs** is set.

s   Forces using single precision. Implies **–Fc**.

S   <u>Default</u>

USAGE

-f

Option:

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

–f *file*

Arguments:

A filename for command line processing. The filename "-" may be used to denote standard input.

Description:

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

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

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

1. It is possible to have multiple arguments on the same line in the command file.

2. To include whitespace in the argument, surround the argument with either single or double quotes.

3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:

   a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.

   b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

Example:

```
"This has a single quote ' embedded"
```

or

```
'This has a double quote " embedded'
```

or

```
'This has a double quote " and \
a single quote '"' embedded"
```

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

Example:

```
"This is a continuation \
line"
        -> "This is a continuation line"

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

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

Example:

Suppose the file **mycmds** contains the following lines:

```
-err
test.c
```

The command line can now be:

c166 -f mycmds

–G

Option:

From the **Project** menu, select **Project Options...**
Expand the **Application** entry and select **Memory Model**.
Select the **Medium** or **Large** memory model.
Expand the **C Compiler** entry and select **Allocation of Variable**.
Enter a name in the **Near data group name** field.

–G*groupname*

Arguments:

The name for a group of near data sections.

Description:

With this option you can specify a name for a group of near data sections. This option can only be used in the medium and large memory model.

See sections 3.2.1.7 *Efficiency in Large Data Models (Medium/Large/Huge)* and 3.12 *Interrupt* for more details.

-g

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Miscellaneous**.
Enable the **Generate high level language debug infomation** check box.
Optionally, enable one or more of the other check boxes.

–g[**b** | **f** | **l** | **s**]

Description:

Add directives to the output files, incorporating symbolic information to facilitate high level debugging. Note: using **–g** may turn off some peephole optimizations.

With **–gb** 'bit' type information and pointer behavior description is omitted for compatibility with old IEEE-695 consuming tools.

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

With **–gl** you disable lifetime information for all types.

With **–gs** user stack adjustment information is omitted for compatibility with old IEEE-695 consuming tools. If you use **–gs** it is also recommended to invoke **ieee166** with the **–c1** option. This combination gives the best compatibility with old IEEE-695 consuming tools. When you invoke the control program **cc166** with **–gs** this will also set **–c1** on invocation of **ieee166**.

Examples:

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

```
c166 –g test.c
```

To add symbolic debug information to the output files but disable lifetime information for all types, enter:

```
c166 –gl test.c
```

USAGE

-gso

Option:

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

–gso
–gso=*file.gso*

Arguments:

The name of a `.gso` file with object allocation information for the final build.

Description:

Enable the global storage optimizer. Please refer to section *gso166* in Chapter *Utilities* of the *Cross-Assembler, Linker/locator, Utilities User's Manual* for more details.

Examples:

```
c166 module.c –gso
```

Generates the file `module.sif` (Source Information File) with information on all global objects.

```
c166 module.c –gso=module.gso
```

Generates `module.c` with the global objects allocated as specified in the `module.gso` file.

-H

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Preprocessing**.
Enter one or more filenames in the **Include these files before source**
field, separated by semicolons.

–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:

```
c166 –Hstdio.h test.c
```

–I

USAGE

-i

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Miscellaneous**.
Select a scaling factor in the **Interrupt vector scale** box. Note that this
item is only available for XC16x/Super10 architectures (ext2).

–i*scale*

Arguments:

Specify scaling of the interrupt vector table.

Description:

The XC16x/Super10 architectures (ext2) allows a scalable interrupt vector
table. This option can be used to specify the scaling factor:

Scale	Factor	Size
0	x1	4 bytes / vector (no scaling)
1	x2	8 bytes / vector
2	x4	16 bytes / vector
3	x8	32 bytes / vector

Table 4-4: Scaling factor

Depending on the size of an interrupt vector table entry, the compiler will
try to place as much code from an interrupt function inside the vector
table as possible.

This option can only be used in conjunction with the **–x2** option.

Example:

```
c166 –x2 –i3 test.c
```

Selects the XC16x/Super10 architectures (ext2) and specifies that each
interrupt vector table entry is 32 bytes in size.

-I

Option:

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

–I*directory*

Arguments:

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

Description:

Change the algorithm for searching #include files whose names do not
have an absolute pathname to look in *directory*. Thus, #include files
whose names are enclosed in "" are searched for first in the directory of
the file containing the #include line, then in 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
C166INC. C166INC 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 C166INC and the relative path) are
still searched.

Example:

```
c166 –I/proj/include test.c
```

Section 4.4 *Include Files*.

USAGE

–M

Option:

From the **Project** menu, select **Project Options...**
Expand the **Application** entry and select **Memory Model**.
In the **Memory model** box, select a memory model.

–M_model_

Arguments:

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

t   tiny (cpu in non-segmented mode)
s   small
m   medium
l   large
h   huge

Default:

–Ms

Description:

Select the memory model to be used.

Example:

```
c166 –Ml test.c
```

Section 3.2.1 _Memory Models_.

-m

Option:

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

–m_mem_**=**_size_

or

–m_mem_**=**[_size_],_threshold_

Arguments:

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

mem	Description	Default size limit (bytes)
BI	bits	2048 (bits)
CO	strings / floating point	none
BA	bitwords	256
NB	near data	none
FB	far data	none
XB	shuge data	none
HB	huge data	none
PR	functions	65536
SB	system data	16384
IR	internal ramdata	2048

Table 4-5: Memory spaces

A threshold value. The default is no threshold.

Description:

Specify the memory _size_ (limits) to be used by the compiler for checking static memory allocations of the module being processed. If the **–t** option is used the size allocated by the module is reported, when **c166** completes compilation.

USAGE

When a section is equal or larger than the threshold size, the compiler will switch to a new selection with the identical attributes and class for subsequent allocations. The threshold size is memory dependent. A size of zero means no threshold and this is the default. Specifying a threshold size is particularly useful when compiling very big modules or when there are too many initiialized variables in a single module.
Example:

    –mPR=0,4000

is suitable for compiling modules with more than 64Kb code without getting too many sections.
Likewise:

    –mFB=0,4000

allows more than 16Kb of initialized far data in a single module by switching to a new section after approximately 4Kb. However, it will result in numbered sections with different names, so it might be necessary to adapt the linker/locator invocation when locator controls refer to a particular section by name.

-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:

```
c166 –misrac9 test.c
```

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

USAGE

-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:

```
c166 –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 but send the output to `stdout`.

This option is for example useful to quickly inspect the output or to redirect the output to other tools.

Example:

```
c166 –n test.c
```

The compiler sends the output (normally `test.src`) to stdout and does not create the file `test.src`.

USAGE

-O

Option:

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

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

–O_flags_

Arguments:

One or more optimization flags.

Default:

–O1

Description:

Control optimization. By default **c166** performs as much code
optimizations as possible (same as **–O1**).

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. An overview of the flags is given below.

a - relax alias checking
b - no clearing of non-initialized static and
       public variables
c - common subexpression elimination
d - data flow, constant/copy propagation
e - allocate (constant) romdata in PDAT instead
       of LDAT (only with **–Ms**)
f - optimize for speed (increases code size)
g - enable expression recognition
h - optimize interrupt frame
j - peephole optimization
k - register contents tracing
l - fast loops (increases code size)
m - instruction reordering

    n - NOP removal
    o - code order rearranging
    p - control flow optimization
    q - use far pointer when converting to/from long
    r - optimize allocation of register variables
    s - use jump table for switch statement
    t - turn tentative into defining occurrence
    u - use user stack for interrupt
    v - data flow analysis peephole (DFAP)
    w - relax alias checking: assume no cross type aliasing
    x - inline the intrinsic version of some C library functions

Example:

```
c166 –OAcdFhkLnprstVw test.c
```

USAGE

-Onumber

Option:

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

–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 **–OABCDEFGHJKLMNOPQRSTUVWX** (no optimization)
–O1 - same as **–OABcdEFghjkLmnopQrS*UVwX** (default)
–O2 - same as **–OaBcdEFghjkLmnopQrS*UVwX** (size)
–O3 - same as **–OaBcdEfghjklmnopQrS*UVwx** (speed)
  * = **t** for **–Mm**/**–Ml**/**–Mh**, **T** for **–Mt**/**–Ms**

Example:

To optimize for code size, enter:

```
c166  –O2 test.c
```

-Oa / -OA

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Relax all alias checking**.

–Oa / –OA

Pragma:

noalias / alias

Default:

–OA

Description:

With **–Oa** you relax alias checking. If you specify this option, **c166** 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, the compiler 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 4.6 *Alias* in this chapter.

Pragmas `noalias` and `alias` in section 4.5, *Pragmas*.

USAGE

-Ob / -OB

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation of Variables**.
Enable or disable the **Perform 'clearing' of of non–initialized
static/public variables** check box.

–Ob / **–OB**

Default:

–OB

Description:

With **–Ob** the compiler performs no 'clearing' of non-initialized static and
public variables.

With **–OB** the compiler performs 'clearing' of non-initialized static and
public variables.

Section 3.9 *Non-Initialized Variables*.
Pragma `noclear` and `clear` in section 4.5, *Pragmas*.

-Oc / -OC

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Common Subexpression Elimination (CSE)**.

–Oc / **–OC**

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 on the user stack.

With **–OC** you disable CSE (common subexpression elimination). With this
option specified, the compiler will not try to search for common
expressions.

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 */
}
```

Pragmas `cse resume` and `cse suspend` in section 4.5, *Pragmas*.

USAGE

-Od / -OD

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Constant and copy propagation**.

–Od / **–OD**

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;
}
```

-Oe / -OE

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation of Variables**.
Enable or disable the **Allocate constant ROM data in near memory**
check box.

–Oe / **–OE**

Default:

–OE

Description:

With **–Oe** you enable allocation of constant romdata 'CROM' in paged data
sections (PDAT). This option is explained in section 3.2.5 *Constant
Romdata Section Allocation*.

With **–OE** standard allocation of constant romdata 'CROM' in linear data
sections (LDAT) is done.

These options only affect the code generation and section allocation in the
small memory model.

Section 3.2.5 *Constant Romdata Section Allocation*.
Pragmas `switch_tabmem_far`, `switch_tabmem_near` and
`switch_tabmem_default` in section 4.5, *Pragmas*.

USAGE

-Of / -OF

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Favor code size above execution speed**.

–Of / –OF

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.

With **–OF** you produce small code. Favour code density above execution speed. If **–OF** is specified, **c166** calls a run-time library routine for a number of operations.

Pragmas speed and size in section 4.5, *Pragmas*.

-Og / -OG

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Code recognition to generate optimal code for expressions**.

–Og / **–OG**

Default:

> **–Og**

Description:

> With **–Og** you enable expression recognition. Expressions for which very efficient code can be generated are recognized and optimal code is emitted.
>
> With **–OG** you disable expression recognition. Handle expressions that could be recognized using the **–Og** option as generic cases.

USAGE

-Oh / -OH

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Optimization of interrupt frame code for C interrupt functions**.

–Oh / –OH

Default:

–Oh

Description:

With **–Oh** you enable optimization of interrupt frame code for C interrupt functions.

With **–Oh** you disable optimization of interrupt frame code for C interrupt functions.

Section 3.12 *Interrupt* in chapter *Language Implementation*.

-Oj / -OJ

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Peephole optimizer (remove redundant code)**.

–Oj / **–OJ**

Default:

–Oj

Description:

With **–Oj** you enable peephole optimization. Remove redundant code.

With **–OJ** you disable peephole optimization.

Optimization option NOP removal **–On**.

-Ok / -OK

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Trace contents of registers for reuse without reloading**.

–Ok / **–OK**

Default:

–Ok

Description:

With **–Ok** you trace the contents of registers and try to reuse the registers without reloading.

With **–OK** you disable register contents tracing.

Example:

```
/*
 * Compile with -OK -O0
 * Compile with -Ok -O0, register contents tracing,
 * one register is reused
 */
int a, c;

void f(register int b)
{
    a = 22;
    if (b)
    {
        c = 22;
    }
}
```

-Ol / -OL

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Generate fast loops (increases code size)**.

–Ol / **–OL**

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. The smallest code is generated for 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();
    }
}
```

USAGE

-Om / -OM

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable the **Instruction reordering** check box.

–Om / **–OM**

Default:

–Om

Description:

With **–Om** you enable instruction reordering for ext2 targets.

With **–OM** you disable instruction reordering for ext2 targets.

The **–OM** option overrules the pragmas reorder/noreorder in the
source. So, **–OM** always disables instruction reordering, no matter the
pragma settings in your source. With **–Om** active (default) you can control
the instruction reordering with the pragmas reorder/noreorder. If none
of these pragmas are present in your source, the default is reorder.

Pragmas reorder and noreorder in section 4.5, *Pragmas*.

-On / -ON

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **NOP removal by peephole optimizer**.

–On / –ON

Default:

–On

Description:

With **–On** you enable NOP removal by peephole optimizer.

With **–ON** you disable NOP removal by peephole optimizer.

USAGE

-Oo / -OO

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Code order rearranging in flow optimization**.

–Oo / –OO

Default:

–Oo

Description:

With **–Oo** you enable code rearranging in flow optimization.. Try to move (sub)expressions to get faster code. Some debuggers may have difficulties with such options.

With **–OO** you disable code rearranging.

-Op / -OP

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Extra flow optimization pass on intermediate representation**.

–Op / **–OP**

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();
    }
}
```

USAGE

-Oq / -OQ

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Convert pointer to/from long as far pointer**.

–Oq / –OQ

Default:

–OQ

Description:

With **–Oq** you treat casting a pointer to long equal to casting a pointer to a far pointer.

With **–OQ** you treat casting a pointer to long equal to casting a pointer to a huge pointer.

-Or / -OR

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Automatic C register variable allocation**.

–Or / –OR

Default:

–Or

Description:

With **–Or** you retrieve better code. Enable automatic C register variable allocation, unless overruled by the **–r**nr option. If you do not want a certain automatic to be allocated in a register (e.g. `setjmp()`/`longjmp()` pair used), you can declare this variable to be volatile and yet still use the **–Or** option!

With **–OR** you disable automatic C register variable allocation.

-Os / -OS

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Use smart approach for switch statement (do not force jump table)**.

–Os / –OS

Default:

–OS

Description:

With **–Os** you force the compiler to generate jump tables for switch statements.

With **–OS** the compiler chooses the best switch method possible, jump chain or jump table. So, with **–OS** a jump table can still be generated.

Example:

```
/*
 * Compile with -OS, generate jump chain.
 * Compile with -Os, generate jump table.
 */
int i;

void
main( void )
{
      switch (i)
      {
            case 1:   i = 0;
            case 2:   i = 1;
            case 3:   i = 2;
            default:  i = 3;
      }
}
```

Section 3.14 *Switch Statement*.
Pragmas switch_force_table and switch_smart in section 4.5, *Pragmas*.

-Ot / -OT

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation of Variables**.
Select an item from the **Tentative declarations** box.

–Ot / –OT

Default:

–Ot   (medium and large model)
–OT   (tiny and small model)

Description:

With **–Ot** the compiler turns tentative declarations (such as 'int i;') into
defining occurrences (e.g. 'int i=0;').

With **–OT** declarations remain tentative as long as possible.

Section 3.2.1.7 *Efficiency in Large Data Models*.

-Ou / -OU

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Use user stack for interrupt functions**.

–Ou / **–OU**

Default:

–OU

Description:

With **–Ou** the compiler uses the user stack instead of the system stack for task switch (interrupt).

With **–OU** the compiler uses the system stack for task switch (interrupt).

-Ov / -OV

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable the **Dataflow analysis peephole (DFAP)** check box.

–Ov / –OV

Default:

–OV

Description:

With **–Ov** you enable the data flow analysis peephole (DFAP) optimizer.

With **–OV** you disable the data flow analysis peephole (DFAP) optimizer.

This optimizer uses data flow analysis in the peephole to optimize the generated code. Unlike the normal peephole, DFAP has function scope and can optimize when there are program flow changes involved.

Note that the use of DFAP may have a performance penalty on the compiler itself. The DFAP optimizations are rather aggressive and can make programs less debugable.

The **–OV** option overrules the pragmas `dfap`/`nodfap` in the source. So, **–OV** (default) always disables the DFAP optimizer, no matter the pragma settings in your source. With **–Ov** active you can control the DFAP optimizer with the pragmas `dfap`/`nodfap`. If none of these pragmas are present in your source, the default is `dfap`.

Pragmas `dfap` and `nodfap` in section 4.5, *Pragmas*.

USAGE

-Ow / -OW

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Relax cross type alias checking**.

–Ow / **–OW**

Default:

–Ow

Description:

With **–Ow** the compiler relaxes alias checking, assuming there are no pointer aliases for different type. For example, when a pointer to an `int` is dereferenced (written), it is reasonable to assume that this cannot have any effect on `char` objects.

With **–OW** the compiler performs cross-type alias checking.

-Ox / -OX

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Optimization**.
Select **Custom optimization** in the **Optimization** box.
Enable or disable **Inlining of some small C library functions**.

–Ox / –OX

Default:

–OX

Description:

With **–Ox** you enable extra inlining of C library functions. It is only worthwhile to inline C library functions which are very small and frequently used. Therefore, only the following C library functions are inlined in small and tiny memory model. Inlining C library functions is not conform the ANSI-C standard. Extra inlining will be disabled when compiling with inlining allowed, see option **–Ai**/**–AI**. Remember that you cannot take the address of an inline function and you cannot define one of these functions yourself when **–Ox** is active.

The next C library functions are inlined for tiny and small memory model:

strcpy(), strlen(), strchr(), strcmp(),
strcat(), memset(), memcpy()

With **–OX** you disable extra inlining of the C library functions mentioned above.

USAGE

-o

Option:

–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 *file* as output filename, 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:

```
c166 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`.

–P

Option:

From the **Project** menu, select **Project Options...**
Expand the **Application** entry and select **Memory Model**.
Enable the **Use user stack for return addresses** check box.

–P[**d**]

Description:

Enable user stack model. See section 3.2.2, *User Stack Model* for details.
Requires linking with user stack model library unless **–Pd** is specified.

Appendix D, *User Stack Model*.

USAGE

-R

Option:

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

–R{**cl**|**co**|**al**}*mem*=*new*

Pragma:

class / **combine** / **align**

Arguments:

mem is a two letter abbreviation indicating the memory area of a C
program. *mem* can be one of:

mem	Description
BI	bits
CO	strings / floating point
BA	bitwords
NB	near data
FB	far data
XB	shuge data
HB	huge data
PR	functions
SB	system data
IR	internal ramdata

Table 4-6: Memory spaces

new is the new class name, combine type or align type for *mem*.

Description:

The compiler defaults to a section naming convention as described in the
section 3.2.3, *Section Allocation*. With this option you can change the class
name, combine type or align type of a compiler generated section for
mem.

In case a module must be loaded at a fixed address or a data section needs a special place in memory, the **–R** option enables you to generate a unique class name, combine type or align type with a section name. With **–Rcl***mem***=***new* you can specify a *new* class name for *mem* (same as pragma **class**). With **–Rco***mem***=***new* you can specify a *new* combine type for *mem* (same as pragma **combine**). With **–Ral***mem***=***new* you can specify a *new* align type for *mem* (same as pragma **align**). In this way the order **l166** allocates these sections can be specified in a locator command file.

Section 3.2.3, *Section Allocation*.
Pragmas `align`, `class` and `combine` in section 4.5, *Pragmas*.

-r

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Miscellaneous**.
Enter the register definition in the **Register usage** field.
Optionally, enter a register name in the **Register bank name** field. If the
bank must be 'common' enable the **Make register bank common** check
box.

-r[*name*[,**c**]][,*regdef*]

Pragma:

regdef

Arguments:

name is the name of the register bank for this module.

c is the common flag.

regdef is the register bank definition.

Description:

With the **-r** option you can specify the name of the register bank, and
optionally if this register bank must be 'common' (**c**) or not.

The *regdef* argument is specified as a comma separated list of register
ranges. A range is defined as:

R*x*[-R*y*]

When a register bank is declared common, the resulting range must
consist of consecutive registers, starting from R0. In all cases R0 must be
present in the register definition. If not, the compiler adds this register and
generates a message. The register definition remains valid until the next
`#pragma regdef` in the source.

When only the *name* and optional common flag are used in the **-r** option,
a full register bank consisting of R0–R15 is the default.

When the **–r** option is used without any arguments, the REGDEF directive for this module will be omitted. Note that register banks originating from the `_using()` function qualifier will still be generated. Interrupt functions that do not have the `_using()` qualifier use the module's register bank. Since this bank will be omitted, no code will be generated in the interrupt frame to switch register banks.

With `#pragma regdef` the used register set can be redefined. A pragma setting will remain active until the next `#pragma regdef`. The syntax for *regdef* is the same as in the **–r** option. As an alternative the number of registers, starting from R0 may be specified.

When different register sets are used for different functions, the compiler will combine the register sets for the same register bank.

When `#pragma regdef` is used without arguments, or with argument 0, the REGDEF directive for interrupt functions will be omitted, even if the `_using()` qualifier is used. In this case the compiler will not generate code in the interrupt frame to switch global register banks.

When the register set that the compiler can use for code generation is limited this can result in larger code. When the register set is too small the compiler may not be able to generate code at all. In this case assertion errors can be expected. This is a known restriction and the register set should be increased prior to reporting the assertion error.

Examples:

```
-rmybank
```

This option declares a register bank with name "mybank". Unless otherwise specified with `#pragma regdef`, the register bank will consist of R0-R15.

```
-r,r0,r3-r5,r10-r15
```

This option causes all functions (non-interrupt and interrupt) to use the registers from the given set.

```
#pragma regdef r0,r1
void _interrupt( 0x10 ) _using( SOMEBANK )
ISR0( void )
{
}
#pragma regdef r0,r2
void _interrupt( 0x11 ) _using( SOMEBANK )
ISR1( void )
{
}
#pragma regdef r0,r3
#pragma regdef
void _interrupt( 0x12 ) _using( OTHERBANK )
ISR2( void )
{
}
```

The functions ISR0 and ISR1 will use register bank SOMEBANK. Function ISR0 will only use registers R0 and R1. Function ISR1 will only use registers R0 and R2. The compiler will combine the used register sets and generates the following REGDEF directive:

```
SOMEBANK REGDEF R0-R2
```

Function ISR2 will only use the registers R0 and R3. Since #pragma regdef without arguments is used, the compiler will not generate a REGDEF directive, nor will code be generated to perform a context switch.

–rcomnbank,c,R0,R1,R2–R5

This option declares a common register bank with the name "comnbank":

```
COMNBANK COMREG R0-R5
```

Section 3.12, *Interrupt*.
Pragma regdef in section 4.5, *Pragmas*.

-S

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Allocation of Variables**.
Enable the **Static allocation of automatics (instead of user stack)**
check box.

–S

Pragma:

static

Description:

All functions of the C module are compiled using static memory for
non-register function automatics. This option can be useful for non
recursive applications.

Section 3.6.1 *Static Approach of Function Automatics*
Pragmas `automatic` and `static` in section 4.5, *Pragmas*.

USAGE

-s

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Output**.
Enable the **Merge C source code with assembly output** check box.

–s [**i**]

Pragma:

source

Description:

Merge C source code with generated assembly code in output file.

When the additional '**i**' sub option is specified, the C source of the include files will also be merged.

Example:

```
c166 –s test.c

      NAME    TEST_C
; test.c     1 int i;
; test.c     2
; test.c     3 int
; test.c     4 main( void )
; test.c     5 {
      PUBLIC  _main
TEST_1_PR    SECTION CODE WORD PUBLIC 'CPROGRAM'
_main  PROC  FAR
```

Pragmas `source` and `nosrouce` in section 4.5, *Pragmas*.

-T

Option:

From the **Project** menu, select **Project Options...**
Expand the **Application** entry and select **Memory Model**.
Select the **Medium** or **Large** memory model.
Expand the **C Compiler** entry and select **Allocation of Variables**.
Enter a size in the **Threshold for automatic near data allocation** field.

–T*size*

or

–T[*size*]**,** *size2*

Arguments:

The maximum threshold size in bytes (*size*). Or the threshold size for initialized variables (*size2*)

Default:

–T256

Description:

With this option you can specify a maximum *size* (threshold) for allocating data in default data sections. This is useful when you want to limit the size of the default data group. You can use this option in the medium and large model only.

Initialized variables have an infinite threshold by default. Unless a threshold is specified by a second argument to the **–T** option, they are always allocated in the default far data sections.

Example:

To allocate values of maximum 128 bytes long in default far data sections, enter:

```
c166 –T128 –Mm test.c
```

Section 3.2.1.7 *Efficiency in Large Data Models (Medium/Large/Huge)*.

USAGE

-t

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Output**.
Enable the **Display module summary** check box.

–t

Description:

With this option the C compiler produces totals (a module summary) on
stdout and writes section information in an output file.

Example:

```
c166 –t test.c
```

```
MODULE SUMMARY

Code size (bytes)              =        8
Constant size (bytes)          =        6
Near data size (bytes)         =        2
Far data size (bytes)          =        0
Huge data size (bytes)         =        0
Shuge data size (bytes)        =        0
System data size (bytes)       =        0
Internal ram data size (bytes) =        0
Bit size (bits)                =        0
Bit addressable size (bytes)   =        0
User stack size (bytes)        =        0
Register bank size (GPR's)     =       16

processed 13 lines at 1331 lines/min
total: tokens=34, symbols=226
```

-U

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Preprocessing**.
Undefine one or more predefined macros by disabling the corresponding
check box.

–Uname

Arguments:

The name macro you want to undefine.

Description:

With this option you can undefine an earlier defined macro *name* as with
#undef. This option is for example useful to undefine predefined macros.

The following predefined ANSI C standard macros cannot be undefined:

__FILE__   "current source filename"

__LINE__   current source line number (int type)

__TIME__   "hh:mm:ss"

__DATE__   "Mmm dd yyyy"

__STDC__   level of ANSI standard. This macro is set to 1 when the
                option to disable language extensions (**–A**) is effective.
                Whenever language extensions are excepted, __STDC__ is set
                to 0 (zero).

Example:

To undefine the predefined macro _C166, enter:

```
c166 –U_C166 test.c
```

–D
Section 3.5, *Predefined Macros*.

USAGE

-u

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Language**.
Enable the **Treat all 'char' variables unsigned** check box.

–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:

```
c166 –u test.c
```

• • • • • • • • •

–V

Option:

–V

Description:

Display version information.

Example:

c166 –V

```
TASKING C166/ST10 C compiler    vx.yrz Build nnn
Copyright years Altium BV       Serial# 00000000
```

-w

Option:

From the **Project** menu, select **Project Options...**
Expand the **C Compiler** entry and select **Diagnostics**.
Enable one of the options **Display all warnings**, **Suppress all warnings**, or **Suppress only certain warnings** and enter the numbers, separated by commas, of the warnings you want to suppress.

–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 extensive warnings 183, 196 and 216.

Example:

To suppress warning 135, enter:

```
c166 file1.c –w135
```

-x

Option:

From the **Project** menu, select **Project Options...**
Expand the **Application** entry and select **Processor**.
From the **Processor** box, select a processor or select **User Defined**.
If you selected **User Defined**, expand the **Processor** entry, select **User Defined Processor** and make your changes.

–x[**1**|**2**|**22**|**d**]

Arguments:

Optional features:

d	ST10 with support for the MAC co-processor
1	C166S v1.0 architecture
2	XC16x/Super10 architecture
22	Enhanced Super10 architecture

Description:

The **–x** option selects the processor architecture.

–x	(default) selects the standard C166 extended architecture as used by the Infineon C16x and STMicroelectronics ST10.
–xd	Selects the standard ST10 extended architecture with MAC co-processor support such as the ST10x272.
–x1	Enables support for the C166S v1.0 architecture.
–x2	Enables support for the XC16x/Super10 architecture, including support for the MAC co-processor. Furthermore, this option automatically enables instruction reordering.
–x22	Enables support for enhanced Super10, such as the Super10M345. This includes support for the MAC co-processor, with extra CoMAC instructions, like CoSHL with rounding. A third local register bank is allowed. Furthermore, this option automatically enables instruction reordering. If this is not wanted, use #pragma noreorder to switch this feature off. (Or use the **–znoreorder** command line option).

USAGE

Example:

To use an Infineon XC167, enter:

```
c166 –x2 file.c
```

Pragma `reorder` in section 4.5, *Pragmas*.

-z

Option:

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

–z*pragma*

Arguments:

A pragma as listed in section 4.5, *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.

Example:

To issue a '#pragma autobita 2' using the command line, enter:

```
c166 –zautobita–2 file.c
```

The '-' between `autobita` and `2` is converted to a space.

Section 4.5, *Pragmas*.

4.4 INCLUDE FILES

You may specify include files in two ways: enclosed in <> or enclosed in "". When an #include directive is seen, **c166** 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:

   c166 ..\..\source\test.c

   UNIX:

   c166 ../../source/test.c

   c166 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 (**c166 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:

   PC:

   c166 –I..\include message.c

   UNIX:

   c166 –I../include message.c

3. Check if the environment variable C166INC 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 C166INC by using a separator character. Instead of using **–I** as in the example above, you can specify the same directory using C166INC:

PC:

```
set C166INC=..\include
c166 message.c
```

UNIX:

if using the Bourne shell (sh)

```
C166INC=../include
export C166INC
c166 message.c
```

or if using the C-shell (csh)

```
setenv C166INC ../include
c166 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 **c166** binary. For example:

PC:

c166.exe is installed in the directory `C:\C166\BIN`
The directory searched for the include file is `C:\C166\INCLUDE`

UNIX:

c166 is installed in the directory `/usr/local/c166/bin`
The directory searched for the include file is
`/usr/local/c166/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 C166INC may or may not be terminated with a directory separator, because **c166** inserts this separator, if omitted.

When you specify more than one directory to the environment variable C166INC, you have to use one of the following separator characters:

PC:

  ; , *space*

e.g. **set C166INC=..\include;\project\include**

USAGE

UNIX:

    : **;** **,** *space*

e.g.   **setenv C166INC ../include:/project/include**

4.5 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 generation process: command line options and keywords (e.g., **near** type variables) in the C application itself. The compiler acknowledges these three groups using the following rules:

Command line options can be overruled by keywords and pragmas. Keywords can be overruled by pragmas. Hence, 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.

Most pragmas have a corresponding compiler option at the command line. When no corresponding option is mentioned here, you can use the **–z** option for this purpose. For example,

#pragma nocustack

can be specified at the command line by entering

–znocustack

When the pragma text consists of multiple tokens, they can be separated on the command line with dashes. For example,

#pragma class *mem=name*

would become

–zclass–*mem=name*

c166 supports the following pragmas:

alias

Default. Same as **–OA** option. Perform strict alias checking. See also section 4.6 *Alias*.

noalias

Same as **–Oa** option. Relax alias checking.

align32

When `#pragma align32` is used:

- long data types are aligned on 32-bit boundaries (not for stack objects)
- struct/union data types are aligned on 32-bit boundaries (not for stack objects)
- the size of a struct/union will always be a multiple of 4 bytes
- the `_packed` keyword overrules the pragma
- struct/union members of type long or bit-field are 32-bit aligned.

Packing of bit-field members will still be done according to the base type (int). Meaning that a bit-field will be aligned to the next 32-bit boundary if the bit-field crosses a 16-bit boundary, or if the bit-field is located at a 16-bit boundary.

```
struct s            /* offset  */
{
    int bf1 : 15;   /*  0 bits */
    int bf2 : 2;    /* 32 bits */
    int bf3 : 2;    /* 34 bits */
    int     : 12;
    int bf4 : 5;    /* 64 bits */
};
```

A bit-field of size 0 will align any struct member to the next 32-bit boundary.

noalign32

Default. 16-bit alignment.

suspend_align32
resume_align32

Whith these pragmas you can temporarily suspend/resume the `#pragma align32`.

asm *[args]*

Insert the following (non preprocessor lines) as assembly language source code into the output file. The inserted lines are not checked for their syntax. The *args* are an interface to the C language. See section 3.11 *Inline Assembly* for details.

asm_noflush

Same as **asm**, except that the peephole optimizer does not flush the code buffer and assumes register contents remain valid.

endasm *[args]*

Switch back to the C language. With the *args* variables can be passed to the C language. See section 3.11 *Inline Assembly* for details.

autobita *threshold*

Move chars, (long) ints and struct/unions which are smaller than or equal to the threshold to bitaddressable memory. The declaration may **not** contain any memory modifiers. The default threshold value is set to zero bytes.

Pointers, arrays and function return values are not moved to bitaddressable memory. Local variables are only moved to bitaddressable memory when declared static or compiled with the **–S** option. See also `bita` in section 3.2.1.9.

autobitastruct *threshold*

Move struct/unions which contain at least one bitfield with length 1 to bitaddressable memory. This only applies for structs/unions which are smaller than or equal to the specified threshold. The declaration may **not** contain any memory modifiers. The default threshold value is set to 4 bytes.

Pointers, arrays and function return values are not moved to bitaddressable memory. Local structs/unions are only moved to bitaddressable memory when declared static or compiled with the **–S** option. See also `bita` in section 3.2.1.9.

automatic

Default. Use stack approach for non register function automatics. Support recursion.

USAGE

static

> Use static memory for non register function automatics. Same as **–S** option. See section 3.6.1 *Static Approach of Function Automatics*.

align *mem=atype*

> Same as **–Ral** option. Use *atype* as align type for section of area *mem*.

class *mem=name*

> Same as **–Rcl** option. Use *name* as class for section of area *mem*.

combine *mem=ctype*

> Same as **–Rco** option. Use *ctype* as combine type for section of area *mem*.

cse suspend
cse resume

> When the CSE optimization is switched on (**–Oc**) then a sequence of

```
#pragma cse suspend
#pragma cse resume
```

> has the effect that expressions in between are not part of the CSE optimization. The pragmas have function scope and do not have any effect unless the CSE optimization is switched on. The CSE optimization for expressions can be switch off in a single function by placing

```
#pragma cse suspend
```

> at the start of the functon body.

custack

> <u>Default</u>. Generate a 'C166_US' section estimating the stack usage of a module.

nocustack

> Suppress the user stack estimation.

clear

> <u>Default</u>. Same as **–OB** option. Perform 'clearing' of non-initialized static/public variables. See section 3.9 *Non-Initialized Variables* for more information.

• • • • • • • • •

noclear

Same as **–Ob** option. No 'clearing' of non-initialized static/public variables. See section 3.9 *Non-Initialized Variables* for more information.

default_attributes

Default. Use default section attributes. See the section 3.2.3 *Section Allocation* for details.

save_attributes

Save the current section attributes. See the section 3.2.3 *Section Allocation* for details about changing section attributes.

restore_attributes

Restore the last saved section attributes. A warning is issued when no section attributes were saved. See the section 3.2.3 *Section Allocation* for details about changing section attributes.

dfap

Default. Enable the data flow analysis peephole. The **–Ov** option must be active for this pragma to work.

nodfap

Disable the data flow analysis peephole.

eramdata

Allocate all non automatic initialized variables in both ROM and RAM. The RAM data section has the class name 'CINITERAM' (unless part of the default data group where all sections must have the same class name). Copy from ROM to RAM at startup (transparent for the user). See section 3.8 *Initialized Variables* for details.

iramdata

Default. Allocate all non automatic initialized variables in both ROM and RAM. The RAM data section has the class name 'CINITIRAM' (unless part of the default data group where all sections must have the same class name). Copy from ROM to RAM at startup (transparent for the user). See section 3.8 *Initialized Variables* for details.

romdata

Allocate all non-automatic variables in ROM only. The ROM data section can have the class names 'CROM', 'CNEARROM', 'CFARROM' or 'CHUGEROM' (unless part of the default data group where all sections must have the same class name). See section 3.8 *Initialized Variables* for details.

fix_byte_write

For all code following this pragma the compiler generates two NOP instructions after each instruction which does a byte write. These instructions are: ADDB, ADDCB, ANDB, CPLB, MOVB, NEGB, ORB, SUBB, SUBCB, XORB. This is a bypass for the erroneous byte forwarding on internal RAM problem. This pragma is equivalent to the new command line option **–BB**.

nofix_byte_write

Default. For all code following this pragma the compiler does not generate two NOP instructions after each instruction which does a byte write. This pragma is equivalent to the new command line option **–Bb**. By default the generation of two extra NOP instructions after a byte write operation is disabled.

The pragmas **fix_byte_write**/ **nofix_bytewrite** and the **–BB** option only have to be used for the steps of the SAB 88C166 (flash), which have the "Erroneous Byte Forwarding for internal RAM locations". Please refer to the Infineon errata sheets of your CPU step for more information.

fragment
fragment resume
fragment continue

Controls fragmentation of code memory. See section 3.2.4, *Code Memory Fragmentation* for details.

global_dead_store_elim

Default. Enable dead store elimination on global and local static variables.

no_global_dead_store_elim

Disable dead store elimination on global and local static variables.

Example:

```
void func (void)
{
    enable=1;
    while ( !activity  );
    enable=0;
}
```

The first assignment will not be optimized away when this pragma was used.

indirect_access [((_)near)]address[-address],...

With this pragma you can specify the address (ranges) that have to be accessed using an indirect addressing mode. When a near address is specified the near or _near keyword must be added, otherwise the address will be interpreted as _huge. In the segmented memory models, _near, _xnear and _system/_iram/_bita are all separate near memory spaces. The correct DPP-number must be included in the address.

Multiple uses of this pragma accumulate the specified addresses.

Example for a segmented memory model:

```
#pragma indirect_access (near)0x9234,(near)0x5232-0x5236,0x200200
/* include DPP number in address */

void main( void )
{
    *((_near int *)0x1234)  = 5;
    *((_xnear int *)0x1234) = 10;
    *((_far int *)0x200200) = 15;
    return;
}
```

This pragma applies to constant pointers only.

m166include "include-file"

This pragma is intended to be used together with user defined intrinsics. This pragma will generate a:

```
$INCLUDE( header.asm )
```

control in the output file. This header file can be used to include the definition of macro (functions) emitted by the compiler when user defined intrinsics are used.

Example:

```
#pragma m166include "header.asm"
```

macro

Default. Perform macro expansion.

nomacro

Do not perform macro expansion.

noframe

Do not emit the interrupt frame code for C interrupt functions. See the section 3.12 *Interrupt* for details.

preserve_mulip

Make the MULIP bit available for use inside interrupt handlers by saving/restoring PSW in interrupt function prologue/epilogue respectively.

public

Default. Public C variables have task scope. See section 3.3 *Task Scope* for details.

global

Public C variables have application scope. See section 3.3 *Task Scope* for details.

regdef [regdef]

See **–r** option and section 3.12 *Interrupt* for details.

reorder

Enable instruction reordering for the XC16x/Super10 architecture (**–x2** option).

noreorder

Disable instruction reordering for the XC16x/Super10 architecture.

• • • • • • • • •

savemac

Save MAC SFRs in an interruptframe. You must use this pragma together with the **–xd**, **–x2** or **–x22** option.

This pragma will not save anything if used together with the **noframe** pragma.

nosavemac

Do not save MAC SFRs in an interrupt frame. You must use this pragma together with the **–xd**, **–x2** or **–x22** option

autosavemac

Save MAC registers in an interrupt frame only when needed. You must use this pragma together with the **–xd**, **–x2** or **–x22** option. If you use this pragma in conjuction with **#pragma noframe**, nothing will be saved.

source

Same as **–s** option. Enable mixing C source with assembly code.

nosource

Default. Disable generation of C source within assembly code.

size

Default. Same as **–OF** option. Favour code density above execution speed.

speed

Same as **–Of** option. Favour execution speed above code density.

stringmem *memory-space*

Controls the allocation of string constants. See section 3.10 *Strings* for details.

switch_force_table

Same as **–Os** option. Allow number of gaps to exceed number of case labels and yet use a jump table. See section 3.14 *Switch Statement* for more details.

USAGE

switch_smart

<u>Default</u>. Same as **–OS** option. Try to use jump table if it is worthwhile. See section 3.14 *Switch Statement* for more details.

switch_tabmem_far

Place jump tables for the small memory model in far ROM. The ROM section where the jump tables are placed have class 'CFARROM'. See section 3.2.5 *Constant Romdata Section Allocation* for details.

switch_tabmem_near

Place jump tables for the small memory model in near ROM. The ROM section where the jump tables are placed have class 'CNEARROM'. See section 3.2.5 *Constant Romdata Section Allocation* for details.

switch_tabmem_default

<u>Default</u>. Jump tables are located as specified by the **–Oe**/**–OE** option. See section 3.2.5 *Constant Romdata Section Allocation* for details.

volatile_union

Treat unions as if declared `volatile`, prohibiting certain optimizations which clash with non-ANSI use of unions: Sometimes a union is used for converting data by writing one member but reading back another.

novolatile_union

<u>Default</u>. Treat unions conform their definition.

4.6   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 alias**). Note that the option **–Oc** must be on to use this option. Relaxing alias checking may reduce code size.

Example 1:

```
int i;

void
func( )
{
   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 */
}
```

USAGE

Example 2:

```
int i;

void
func( 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   */
}
```

Example 3:

```
int array[2];

main()
{
   array[0] = 1;
   array[1] = -1;

   array[0] = array[0] + array[1];
             /* an interrupt might have changed the value    */
             /* of 'array', so 'array' may not be used       */
             /* from register contents, but MUST be read     */
             /* from memory                                  */
             /* --> alias checking MUST be ON in this case   */
}
```

4.7   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, (**c166**'s actual limits are given within parentheses):

Most of the actual compiler limits are determined by the amount of free memory in the host system. In this case a 'D' (Dynamic) is given between parentheses. Some limits are determined by the size of the internal compiler parser stack. These limits are marked with a 'P'. Although the size of this stack is 200, the actual limit can be lower and depends on the structure of the translated program.

- 15 nesting levels of compound statements, iteration control structures and selection control structures (P > 15)
- 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 (P > 31)
- 32 nesting levels of parenthesized expressions within a full expression (P > 32)
- 31 significant characters in an external identifier (full ANSI-C mode),
  500 significant characters in an external identifier (non ANSI-C mode)
- 511 external identifiers in one translation unit (D)
- 127 identifiers with block scope declared in one block (D)
- 1024 macro identifiers simultaneously defined in one translation unit (D)
- 31 parameters in one function declaration (D)
- 31 arguments in one function call (D)
- 31 parameters in one macro definition (D)
- 31 arguments in one macro call (D)
- 509 characters in a logical source line (1500)
- 509 characters in a character string literal or wide string literal (after concatenation) (1500)

USAGE
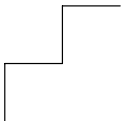
- 8 nesting levels for **#include**d files (50)
- 257 case labels for a switch statement, excluding those for any nested switch statements (D)
- 127 members in a single structure or union (D)
- 127 enumeration constants in a single enumeration (D)
- 15 levels of nested structure or union definitions in a single struct-declaration-list (D)

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.

USAGE

CHAPTER

5

**COMPILER
DIAGNOSTICS**

TASKING

CHAPTER

5

5.1 INTRODUCTION

c166 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 and 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.

If a (non fatal) user error occurs during compilation, **c166** 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 front end. However, in this case, **c166** displays the line number causing the error before the error message. **c166** always generates the error number in the assembly output file, exactly matching the place where the error occurred.

For example, the following program causes a code generator error message:

```
bit b;

void
err()
{
    b = 1;          /* OK */
    b += 1;         /* Not allowed */
}

test.c: 8: }
E 539: (line 7) '+=' not allowed on bit type
```

The output file contains:

```
       PUBLIC      _err
TEST_1_PR          SECTION  CODE WORD PUBLIC 'CPROGRAM'
_err  PROC  NEAR
      BSET  _b
      ERROR C166_ERROR_539
      RET
_err  ENDP
TEST_1_PR  ENDS
```

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 **–c** 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**[*number*] option.

The last class of messages are the internal compiler errors. The following format is used:

S *number*: assertion failed – 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 small C program causing the error.

5.2 RETURN VALUES

c166 returns an exit status to the operating system environment for testing.

For example,

in a BATCH-file you can examine the exit status of the program executed with ERRORLEVEL:

```
c166 –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:

```
c166 $*
case $? in
0)     echo ok ;;
1|2|3) echo error ;;
esac
```

The exit status of **c166** 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

or if the -exit commandline option was used:

0   Compilation successful, no errors/warnings
1   There were user errors/warnings, 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   system 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.

DIAGNOSTICS

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

This error is generated when attempting to cast an object to a function type as shown in the example below:

```
int i;
void main(void)
{
  i+=(int*(int))i;
   return;
   }
```

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

After the **–z** option you must specify an existing pragma. See the description of the **–z** option for details.

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 */
```

Note that this warning is formally correct, but not very useful is most cases. Consider the following situation:

```
unsigned int a;
a = 0x1234u & ~0x00FFu;
```

When you compile this example with option **–Au** (default), this warning appears. Here the type of the unary '~' operator is `int`. Constant folding optimizes the expression to:

```
a = 0x1234u & 0xFF00;   /* right operand is a
                           signed int */
```

Next, the right operand needs to be converted to an `unsigned int` in order to compute the result of the bitwise AND. It is this conversion that generates the warning, since the sign bit is differently interpreted after the conversion.

In most cases you can safely switch off this warning (**–w196** or **–wstrict**).

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

The character set of C source programs is contained within seven-bit
ASCII, but is a superset of the ISO 646-1983 Invariant Code Set. In
order to enable programs to be represented in the reduced set, all
occurrences of the following trigraph sequences are replaced by the
corresponding single character. This replacement occurs before any
other processing.

```
??=  represents  #
??/  represents  \
??'  represents  ^
??(  represents  [
??)  represents  ]
??!  represents  |
??<  represents  {
??>  represents  }
??-  represents  ~
```

The compiler issues a warning when it performs a trigraph
replacement to inform that something occured which was probably not
expected to occur.

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 ”*name*” - 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.

A list of rule numbers is required after the **–misrac** option.

W 208   unsupported MISRA C rule number *number*.

Specified MISRA C rule number is not supported.

E 209   MISRA C rule *number* violation: *rule*

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 occurence of #pragma asm or asm.noflush.

E   215   "pragma endasm" without a "#pragma asm"

W   216   suggest parentheses around assignment used as truth value

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.

Backend

W 501   initializer was truncated

Some most significant bits are non–zero. Due to a cast, the most significant bits are stripped off.

F 504   allocation of *data-type* exceeds *limit*K (memory: *memory*)

A memory overflow occurred. Use a larger memory model or specify a larger storage type. *data-type* can be one of "data", "automatic data" or "code". When *memory* is "program", then try to split the module into separate ones on function basis. It is usually sufficient to split the module into two separate ones, each having about the half of the program code of the original module. Program code of a single function is limited to 64K.

E 519   no indirection allowed on bit type

Pointer to a bit variable and array of bit is **not** allowed, because the 80166 has no instructions to indirectly access a bit variable.

E 531   restriction: impossible to convert to '*type*'

The structure or union cannot be casted to types `bit`, `char`, `int`, `long`, `float` or `double`.

E 539   *operator* not allowed on bit type

See section 3.4.3, *The Bit Type*, for a list of operators that are allowed on type `bit`.

E 540   bit type parameter not allowed

A `bit` type variable is **not** allowed as parameter. The allowed classes for `bit` are: static, public or extern. See also section 3.4.3, *The Bit Type*.

E 541   bit type switch expression not allowed

A `bit` typed expression is **not** allowed as switch expression. See also section 3.4.3, *The Bit Type*.

E 542   argument *number* is not an integral constant expression

The argument of the specified intrinsic function must evaluate to an integral value. See section 3.17, *Intrinsic Functions*, for the syntax of the specified intrinsic function.

DIAGNOSTICS

W 543 'extern near' might be in other data group: check '*Ggroupname*' option is also used with module defining external

If you use the **–G** option, it is your own responsibility to declare 'extern near' variables within the same group. See also section 3.2.1.7, *Efficiency in Large Data Models*.

E 544 semaphore must be bit object

The intrinsic functions `_testset()` and `_testclear()` must have a `bit` type argument.

E 545 maximum interrupt number is 127

Use an interrupt number less than 128.

E 547 calling an interrupt routine, use '_int166()'

An interrupt function cannot be called directly, you must use the intrinsic function `_int166()`.

E 549 argument *number* is not bitaddressable

The intrinsic functions `_getbit()`, `_putbit()` and `_bfld()` require a bitaddressable argument. See section 3.17, *Intrinsic Functions*, for the syntax of these intrinsic functions.

E 550 assignment/parameter/return not allowed with bit-structure

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 as parameter or return value of a function.. See also section 3.4.3, *The Bit Type*.

F 551 too many sections (> *number*)

A module can contain 255 sections at the most.

E 552 '*memory_type*' is illegal memory for function: near or huge only

The specified storage type is not valid for this function. The storage type of a function can be either **near** or **huge**. A function can also have return type `bit`.

F 553 illegal memory model

See the compiler usage for valid arguments of the **–M** option.

F 554 illegal memory type specified

See the description of the **–m** option for the correct syntax.

• • • • • • • • •

F   555   invalid *option* option

The option must have a valid argument. See the description of the *option* for the correct syntax.

F   556   illegal section qualifier in -R option

See the description of the **–R** option for the correct syntax.

F   557   illegal number in option

You must specify a valid number (decimal or hexdecimal) to the option.

W   558   maximum number of GPR's in a registerbank is 16 - ignored

If you specify a number of GPRs to the **–r** option or `#pragma regdef` it must have a value in the range 6-16 (inclusive).

E   560   static initialization of sfr/sfrbit esfr/esfrbit is not allowed

For example, the construction `sfr SYSCON = 2;` is not allowed.

E   561   illegal storage class for sfr/sfrbit, esfr/esfrbit, xsfr

[e]sfr/[e]sfrbit/xsfr is not allowed as static, extern, automatic, register or parameter.

E   562   it is not allowed to change the align type for internal ram data sections

Internal ram data sections are always IRAM addressable.

E   563   ”*function*()”: 0 is invalid interrupt number, use ”main()”

An interrupt number must be in the range 0 to 127 or -1.

E   564   section ”*name*” may not be BYTE aligned

The sectoin must be word, page, segment or PEC aligned.

F   565   Illegal combine type

The combine type must be one of **L** (local), **P** (public), **C** (common), **G** (global), **S** (Sysstack), **U** (Usrstack) or **A** *address* (absolute section AT constant *address*).

F   566   Illegal align type

The combine type must be one of **B** (byte), **W** (word), **P** (page), **S** (segment), **C** (PEC addressable) or **I** (IRAM addressable).

DIAGNOSTICS

E   568   more than 16K initialized data for '*name*': use 'shuge' or use the
          -m option
          more than 64K initialized data for '*name*': use 'huge' or use the
          -m option

Declare explicitiy initialized variables in shuge or huge memory when
the total size of those variables in a module exceeds 16K or 64K
respectively. An alternative is to omit the initializer and to initialize the
variable at run-time as far as needed. `cstartx.asm` clears variables
without explicit initializer automatically.

–m option

E   569   far/huge not allowed in tiny memory model

The `far`, `huge` and `shuge` keywords are not possible (and not
allowed) in the tuny memory model, because all normal data is
implicitly `near`.

E   570   allocation single data object exceeds 16K: use 'shuge'
          allocation single data object exceeds 64K: use 'huge'

Variables greater than 16K or 64K must be declared 'shuge' or 'huge'
respectively.

F   571   '*memory*' is illegal memory for #pragma romdata:
          near/far/huge only

You can only use the `near`, `far`, `huge` and `shuge` keywords on
romdata sections.

W   572   invalid option for this model: '*option*' - ignored

The **–G***groupname* and **–T***size* options are only allowed in the medium,
large or huge memory model.

W   573   conversion of long address to short address

This warning is issued when pointer conversion is needed, for
example, when you assign a `huge` pointer to a `near` pointer.

W   575   c166 language extension keyword used as identifier

A language extension keyword is a reserved word, and reserved words
cannot be used as an identifier.

F   577   -x*char* is invalid suboption

See the description of the **–x** option for the correct syntax.

E   579   'offset' must be a constant value between 0 and 15

The bit offset used in **_atbit** must be a constant value between 0 and 15 (the bit position in an integer).

E   580   REGDEF is too small for register arguments/parameter of
           "*name*": use 'stackparm'

The number of registers is too small for parameter passing. Pass the arguments over the user stack. You can use the **stackparm** keyword for this purpose.

W   582   REGDEF R0-R5 is minimum registerbank

If you specify a number of GPRs to the **–r** option or **#pragma regdef** it must have a value in the range 6-16 (inclusive).

F   583   -F*char* is invalid suboption

See the description of the **–F** option for the correct syntax.

W   585   duplicate function qualifier - '*name* (*number*)' ignored

Only one function qualifier is allowed. The number within parentheses indicates which of the qualifiers is ignored, 0 being the first occurrence.

W   586   duplicate function qualifier - '*name*'

Only one function qualifier is allowed. The duplicate qualifier is ignored.

W   587   '*number*' illegal interrupt/bank number (*min* to *max*) - ignored

An interrupt number must be in the range 0 to 127 or -1. A register bank number must be in the range 1 to 255.

W   588   '*name1*' not allowed with '*name2*' or '*name3*' - ignored

This is an illegal function qualifier combination. Functon qualifier *name1* is ignored.

E   589   interrupt function must have void result and void parameter list

A function declared with **interrupt(*n*)** may not accept any arguments and may not return anything.

E   590   bank function qualifier allowed in small/large/huge model only
           (code >64K)

The **bank(*n*)** function qualifier cannot be used in the tiny and medium memory models. It is only allowed in the small, large or huge memory model. See also section 3.18, *Code Memory Banking*.

E   591   conflict in '*name*' attribute

The attributes of the current function qualifier declaration and the previous function qualifier declaration are not the same.

E   592   different '*name*' number

The function prototype of an interrupt service routine must have the same vector number and using numbers as in the function definition. The same applies to the bank number of a banked function.

W   593   function qualifier used with non-function

A function qualifier can only be used on functions.

E   595   bank function qualifier not allowed with near function

Code memory banking is only useful in the small, large and huge memory model (code > 64Kb).

W   596   #pragma switch_force_table (-Os) ignored: jump table would exceed 16K

The jump table does not fit in 16K.

E   597   indirect near call to function "*function()*" from huge function is not allowed
near call to run-time library function "*function()*" from huge function is not allowed

A huge function may not call any standard C (or run-time) library function, or any other 'near function' in the first segment.

E   598   invalid number atomic instructions, atomic range is [1..4]

The `_atomic` intrinsic function only accepts a number in the range [1..4].

W   599   nothing to restore, no section attributes are saved with #pragma save_attributes

Pragma `restore_attributes` was used without a previous pragma `save_attributes`.

F   602   corrupt initialized variable: different size between initialized RAM and ROM section

The initialized RAM and ROM sections must have the same size. This may be due to a different level of indirection with an assignment.

W 604   possible un-aligned access on byte-label '*name*'

Characters are not aligned. Functions and pointers are always aligned.

E 605   _atbit() only possible on objects, not on constant addresses

Use **_atbit()** to define bit variables within a **bitword** or **sfr** variable with a previously defined name.

E 606   _atbit() only possible for bit/sfrbit objects

Only **bit** and **sfrbit** objects can be declared with **_atbit()**.

E 607   _atbit() only possible on bitword/sfr objects

_atbit() only accepts **bitword** or **sfr** objects as an argument.

E 608   specified object not BIT-addressable

The object specified to **_atbit()** must be a **bitword** or **sfr** object.

E 610   sfrbit object can only have _atbit() on sfr object
            bit object can only have _atbit() on a bitword object

You cannot specify a **sfrbit** object with **_atbit()** on a **bitword** object, and you cannot specify a **bit** object with **_atbit()** on a **sfr** object.

E 611   missing #pragma endasm

You cannot specify a #pragma **asm** or **asm_noflush** when inline assembly is already active. You have to use #pragma **endasm** first.

E 612   missing #pragma asm

The #pragma **endasm** was found while inline assembly was not active. Remove the pragma or insert a #pragma **asm**.

E 613   '(' missing in inline assembly pragma

Check the syntax of the pragma **asm**/**endasm**. '(' was expected. See section 3.11, *Inline Assembly*, for the correct syntax.

E 614   ')' missing in inline assembly pragma

Check the syntax of the pragma **asm**/**endasm**. ')' was expected.

E 615   illegal character '*character*' in inline assembly pragma

Check the syntax of the pragma **asm**/**endasm**. A '=' or '@' was expected.

E   616   illegal pseudo register in inline assembly pragma

A pseudo register name has the following synopsis: @[**w**|**b**|**i**]*num*. See section 3.11, *Inline Assembly*, for more information.

E   617   pseudo register "@*number*" already defined

A pseudo register cannot be defined twice. Use another name or number.

E   618   illegal variable name in inline assembly pragma

The variable name specified after a pragma `asm`/`endasm` is not a valid identifier.

E   619   "*name*" undefined in inline assembly pragma

A C variable with the name you specified to a pragma `asm`/`endasm` does not exist. Check if you specified the correct variable name.

E   620   pseudo register "@*number*" undefined

The pseudo register must first be defined after a pragma `asm`.

E   621   no registers anymore for "@*name*"

There were no free registers left to allocat this pseudo register.

E   622   improper use of "bita"/"bitword" in declaration of "*name*"

The `bita` keyword is only allowed on structures, unions and integral types.

W   720   -OZ no longer supported

This version of the compiler no longer supports the **–OZ** option.

E   724   _at() requires a numerical address

You can only use an expression that evaluates to a numerical address.

E   725   _at() address out of range for this type of object

The absolute address is not present in the specified memory space.

E   726   _at() only valid for global variables

Only global variables can be placed on absolute addresses.

E   727   _at() only allowed on non-initialized variables

Absolute variables cannot be initialized.

• • • • • • • • •

W   728   _at() has no effect on external declaration

When declared **extern** the variable is not allocated by the compiler.

W   729   _at() cannot be used on struct / union members (ignored)

E   730   _at() cannot be used on bit, bita, system, sfr, esfr, xsfr and iram

E   731   _at() this type of object must be word aligned

E   732   _at() address out of range for this memory model

The absolute address does not fit in the specified memory model. You might want to use a larger memory model.

E   733   bad argument to #pragma cse, expect a number, "suspend" or "resume"

See the description of pragma **cse** for more information.

E   734   #pragma cse suspend/resume has no effect outside function body

Pragma **cse suspend/resume** has a function scope.

W   735   pointer conversion restricts arithmetic precision and alignment

When a huge pointer is converted to an shuge pointer, it may lead to incorrect code when the (huge) object it points to crosses a segment boundary. After the conversion, the compiler assumes that the object is 64Kb at most and won't cross a segment boundary. Both assumptions may be wrong. A similar problem arises when converting a shuge or huge pointer to a far pointer. Far objects are limited to 16Kb and never cross a page boundary.

E   736   function *"name"* too big (should be <= 64Kb code)

Break the function into smaller ones.

E   737   function *"name"* doesn't fit in section, try -mPR=0,4000

See the description of the **–m** option for additional information.

W   739   ormask: 0x*hexnumber* does not fit into andmask: 0x*hexnumber*

When the set bits in the ormask do not overlap the set bits in the andmask, these bits might be unintentionally set.

F   740   **–s***char* is invalid suboption

Only '**i**' can be used as a suboption. See the description of the **–s** option for additional information.

E   744   bad #pragma m166include syntax

An error occured when defining a macro-processor include file.

E   745   no registers left for *expression*

There were no free registers left to pass *expression* to a user defined intrinsic.

E   750   _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.

E   752   _localbank qualifier only allowed with interrupt functions

You can only use the `_localbank` function qualifier in combination with the `_interrupt` function qualifier.

W   753   '*name*' not allowed with '*name1*', '*name2*' or '*name3*' - ignored

For example, the `_localbank` function qualifier cannot be used in combination with `_stackparm`, `_bank` or `_using` - ignored.

E   754   *name* function qualifier can only be used in combination with
            -x2

The `_localbank` and `_stacksize` qualifiers can only be used with the XC16x/Super10 architecture.

E   758   _stacksize qualifier only allowed with interrupt functions using a
            local register bank

For example the following is not allowed:

```
void _interrupt(0x10) _localbank(0) _stacksize(20) ISR(void);
```

Because _localbank(0) indicates a global register bank.

W   759   stack size must be even - ignored

The value of the `_stacksize` function qualifier must be even.

W   760   negative stack size adjustment exceeds user stack size
            estimation, truncated

Suppose the compiler estimates that the occupied stack space for an interrupt function is 12 bytes. If '`_stacksize( -14 )`' is added to the function definition, this warning is generated and the value of the `_stacksize` qualifier will be adjusted to -12.

● ● ● ● ● ● ● ● ●

W 761   keyword '*name*' only allowed in combination with **–x2** -
          ignored

The used keyword is only valid for the XC16x/Super10 architecture and
will be ignored if this chip is not selected. (Use   **–x2**)

E 762   option -i can only be used in combination with **–x2** - ignored

E 763   _cached qualifier only allowed with interrupt functions

W 764   #pragma *name* only allowed in small memory model with
          extended instruction set

E 766   initialized ramdata sections don't support section attributes

E 771   variable argument list not allowed with intrinsic function:
          "*name*()"

W 775   obsolete option **–Ff/–FF** - floating point library is reentrant by
          default

The **–Ff** / **–FF** option is no longer needed,

W 781   _at ( ) has no effect on zero sized. object: "%s"

e.g. int a[ ] _at (0x1234);

E 785   _xnear only allowed in medium/large/huge memory model

In the medium/large/huge memory model, the _xnear keyword allows
you to allocate variables in DPP1 which shares this page with the user
stack. In the tiny/small memory model the user stack is located in
_near memory where normal data is also located. Hence this memory
space is already shared. Therefore there is no need for an _xnear
memory space in the tiny/small memory model.

F 787   bad argument in -gso option : *argument*

The syntax of the **–gso** option is -gso=*file.gso* where *file.gso* is the
name of a `.gso` file.

E 788   GSO file not generated by 'gso166'

Missing $GSO166 directive in the `.gso` file.

E   789   GSO file memory model mismatch

$MODEL(*modelname*) in the `.gso` file does not match the compiler memory model.

E   790 - E 849   Reserved for gso166 errors.

E 000 from gso166 maps on compiler error E 790;
E 001 from gso166 maps on compiler error E 791;

etc.

F   850   cannot find object *object* in GSO file

The name of a global object cannot be found in the `.gso` file for automatic storage assignment.

W   851   -T option cannot be used in conjunction with -gso

When you use **gso166** for building the application, **gso166** will assign storage to global objects. However, with the **–T***size* option the compiler is not allowed to allocate global objects in _near memory that exceed the specified *size*.

W   852   pragma *name* cannot be used in conjunction with -gso

You cannot use pragmas that control the storage of global objects in conjunction with **gso166**.

W   860   pragma *name* has no effect inside a function - ignored

You cannot use this pragma inside a function body, use the pragma before or after a function.

W   861   illegal memory space in pragma *name* - ignored

See section 3.10 *Strings* for a list of all available memory spaces with `#pragma stringmem`.

W   862   bad argument to pragma *name* - ignored

See section 3.10 *Strings* for a list of all available arguments of `#pragma stringmem`.

W   864   _atbit() has no effect on external declaration

Do not use _atbit() on external declarations of a bit object. Use _atbit() on the definition instead.

E   865   object: '*name*' containing bit-type fields is forced into
                bitaddressable area

This error occurs when you use a storage type for a structure with
bit-type members.

E   866   pointer: '*name*' is forced to bitaddressable, pointer to
                bitaddressable is illegal

A pointer to bitaddressable memory is not allowed.

E   876   cannot initialize _atbit() object, initialize base object '*name*'
                instead

Global bits declared with the _atbit() attribute cannot be initialized.
Initialization should be done on the base object instead.

E   877   cannot generate code for multiple architectures

You specified more than one processor architecture to the **–x** option,
for example **–x12**. Specify only one processor architecture.

W   878   obsolete option *name*; replacement option -x1 activated

The silicon bug workaround controls c166sv1sp, c166sv1sp2 and
c166sv1sp are no longer used. Specify the extend1 processor
architecture (**–x1**) instead to activate all three silicon bug workarounds.

W   879   obsolete option *name* ignored

The specified option is no longer in use. It may disappear in a future
version of the compiler, resulting in a command line syntax error.

W   880   *class* '*object-name*' is always aligned

The object can never be located at an odd address. Despite this, the
compiler sometimes generates code to access the object as if it were
unaligned. This will lead to an unneccessary increase of code size.
Therefore, you should remove the `_noalign` qualifier when this
warning is generated. Whether or not the compiler generates unaligned
proof code is undefined in this case.

Sample warnings:

```
automatic/parameter 'p' is always aligned
return value of 'func' is always aligned
```

DIAGNOSTICS

E   881   register R*n* is outside REGDEF definition and is needed for code
            generation

This error may occure when the size of the register bank has been
decreased with `#pragma regdef` or the **–r** command line option and
a specific register outside the defined register bank is needed for code
generation. The need for a specific register may arise in situations as
listed below:

1. function return-values (R4-R5)

2. when handling function parameters (R12-R15)

3. run-time and floating-point library calls

Situation 1. should never cause an error because a minimum register
bank always includes R4-R5. You can avoid situation 2. by using the
`_stackparm` function qualifier. This will force all parameters on the
user stack. In all other situations the size of the register bank needs to
be increased.

F   882   common register bank can consist of one range only and must
            start with R0

Adjust the register bank definition accordingly. See the **–r** option for
the correct syntax.

F   883   illegal register bank definition: *regdef*

See the **–r** option for the correct syntax.

W   884   common register bank can consist of one range only and must
            start with R0 -- extended

Adjust the register bank definition accordingly. See the **–r** option for
the correct syntax.

W   885   R0 not included in register bank definition -- forced

The compiler adds register R0 to the register bank definition.

W   886   illegal register bank definition: *regdef*

The illegal register bank definition is ignored. See the **–r** option for the
correct syntax.

W   887   option **–x***c* is deprecated -- implied by default

The **–x** option is always on by default.

• • • • • • • • •

W 888   struct/union member cannot have an explicit memory specifier
          -- ignored

An individual struct/union member cannot be allocated in a specific
memory space. It is only possible to allocate a complete struct/union in
a particular memory space.

For the example below, the warning will be generated:

```
struct s
{
    _huge int member;
};
```

The correct way to add the memory specifier is:

```
struct s
{
    int member;
};

_huge struct s st_huge;   /* struct s in _huge memory */
_near struct s st_near;   /* struct s in _near memory */
```

E 892   #pragma indirect_access: address expected

An address was expected.

E 893   #pragma indirect_access: invalid address: *address*

An invalid address has been specified.

E 894   #pragma indirect_access: first address must be lower than last
          address

When an address range is specified, the first address must be lower
than the last address.

E 895   #pragma indirect_access: not a near address: 0x*hexnumber*

The specified address cannot be near. e.g:

```
#pragma indirect_access (near)0x1234
```

when translated in a segmented memory model, this is not a valid near
address, because DPP0 is never used to address a near memory space.

CHAPTER 6

LIBRARIES

TASKING

6.1 INTRODUCTION

c166 comes with libraries 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).

Four sets of libraries are delivered to meet specific requirements for the various C16x/ST10, XC16x/Super10 microcontroller architectures. These sets are located in separate directories:

ext   The extended libraries are needed for the C16x/ST10 and similar architectures. These architectures feature the extended instruction set, extended special function registers, 24-bit addressing and extended PEC pointers. Use these libraries in conjunction with the compiler option **–x** or **–x1**.

extp   The protected libraries provide a software workaround for CPU functional problems. Use these libraries in conjunction with the compiler options **–x** or **–x1** and **–B**.

ext2   The extended 2 libraries are needed for the XC16x/Super10 and similar architectures. These architectures feature jump prediction, scalable and relocatable interrupt vector table, local register banks and instruction reordering. Use these libraries in conjunction with the compiler option **–x2**.

ext2p   The protected libraries provide a software workaround for CPU functional problems. Use these libraries in conjunction with the compiler options **–x2** and **–B**.

Another four sets of libraries are delivered to meet specific User Stack Model requirements for the various microcontroller architectures. These libraries must be used in conjunction with the additional compiler option **–P**. These sets are located in separate directories:

uext   The User Stack Model variant of the extended non-protected libraries.

uextp   The User Stack Model variant of the extended protected libraries.

uext2   The User Stack Model variant of the extended XC16x/Super10 architectures non-protected libraries.

uext2p   The User Stack Model variant of the extended XC16x/Super10 architectures protected libraries.

• • • • • • • • •

Each library set contains the following libraries:

c166?[s].lib   C library. The optional [s] stands for single precision floating point (all floating point arithmetic is in single precision instead of ANSI double precision).

fp166?[t].lib

Floating point library. The optional [t] stands for trapping floating point (using boundary checking and the floating point trap mechanism).

rt166?[s][m].lib

Run-time library. The optional [s] stands for single precision floating point. The optional [m] stands for MAC optimized (use MAC instructions in some basic operations for optimization).

The question mark '?' in these library names must be replaced by a letter representing the selected memory model:

    t   tiny
    s   small
    m   medium
    l   large
    h   huge

All C library functions are described in the section *C Library Interface Description*. These functions are only called by explicit function calls in your application program. However, some compiler generated code contain calls to run-time library functions that would use too much code when generated as inline code. The name of a run-time library function always contains two leading underscores. For example, to perform a long (32 bit) signed division, the function **__sdil** is called.

Because **c166** generates assembly code (and not object code) it adds a leading underscore to the names of (public) C variables to distinguish these symbols from 80166 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. Therefore, you should avoid names starting with an underscore. Note that 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.

LIBRARIES

The code sections of the C166 library have the class 'CLIBRARY', 'SHAREDCLIB', 'RTLIBRARY' or 'SHAREDRTLIB' allowing the library to be allocated in a special memory area via the CLASSES control of **l166**.

6.2 SMALL, MEDIUM AND LARGE I/O FORMATTERS

The C library contains the SMALL I/O formatter version of the printf() and scanf() functions and their variants like sprintf(), fprintf(), etc. This SMALL version does not contain the required functionality to handle precision specifiers and floating point I/O which can specified in the format argument of these functions.

The following extra libraries are included to support easy switching between the three I/O formatter versions:

MEDIUM I/O formatter library no floating point I/O supported precision specifiers supported `fmtio?m.lib`.

LARGE I/O formatter library floating point I/O supported precision specifiers supported `fmtio?l[s].lib`.

The question mark '?' in these library names must be replaced by a character representing the selected memory model:

t   tiny
s   small
m   medium
l   large
h   huge

These I/O formatter libraries are included in all library sets. You can use the control program options **–libfmtiom** and **–libfmtiol** to select the MEDIUM and LARGE I/O formatter libraries.

If no **cc166 –libfmtio*** option is specified on the commandline, then the SMALL printf()/scanf() formatter variant is linked from the C library.

In EDE you can select an I/O formatter library as follows:
From the **Project** menu, select **Project Options...** Expand the **C Compiler** entry and select **Libraries**. Select a **Printf() and scanf() I/O formatters** option.

6.3 SINGLE PRECISION FLOATING POINT

In ANSI C all mathematical functions (`<math.h>`), are based on `double` arguments and `double` return type. So, even if you are using only `float` variables in your code, the language definition dictates promotion to `double`, when using the math functions or floating point formatters (`printf()` and `scanf()`). The result is more code and less execution speed. In fact the ANSI approach introduces a performance penalty.

To improve the code size and execution speed, the compiler supports the option **–F** to force single precision floating point usage. If you use **–F**, a `float` variable passed as an argument is no longer promoted to `double` when calling a variable argument function or an old style K&R function, and the type `double` is treated as `float`. It is obvious that this affects the whole application (including libraries). Therefore special single precision versions of the floating point libraries are now delivered with the package. When using **–F**, these libraries must be used. It is not possible to mix C modules created with the **–F** option and C modules which are using the regular ANSI approach.

For compatibility with the old **–F** option, the **–Fc** option is introduced. This option only treats floating point constants (having no suffix) as `float` instead of `double`.

In EDE you can set floating point options as follows:
From the **Project** menu, select **Project Options...** Expand the **C Compiler** entry and select **Floating Point**. Enable or disable floating point options.

6.4 CAN SUPPORT

The Infineon CAN protocol driver software routines including pre-built CAN libraries are supplied with the 32-bit Windows 95/98/NT version of this product. The file `ap292201.pdf` describes the usage of these libraries. This file is located in the `doc/pdf` directory.

See section 6.7, *CAN Library Interface Description*, for a description of the CAN library routines.

The can166?.Lib CAN libraries are available for all memory models in the `ext`, `extp`, `uext` and `uextp` library sets. These libraries can be rebuilt using the corresponding makefiles.

LIBRARIES

6.5   HEADER FILES

The following header files are delivered with the C compiler:

<**assert.h**>   assert

<**c166.h**>   Special file for portability between **c166** and other C
compilers. Contains macros to enable or disable the usage of
TASKING C166/ST10 language extensions.

<**can_ext.h**>
CAN libraries function prototypes: check_busoff_16x,
check_mo_16x, check_mo15_16x, def_mo_16x, init_can_16x,
ld_modata_16x, rd_modata_16x, rd_mo15_16x, send_mo_16x

<**canr_16x.h**>
Definitions of CAN module control registers. 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.

<**fcntl.h**>   open. Also contains definitions of flags used by _open().

<**float.h**>   isinf, isinff, isnan, isnanf. Constants related to floating point
arithmetic.

<**fss.h**>   Definitions for file system simulation.

<**iso646.h**>
Alternative spellings. No C functions.

<**limits.h**>   Limits and sizes of integral types. No C functions.

<**locale.h**>   localeconv, setlocale. Delivered as skeletons.

<**math.h**>   acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod,
frexp, hypot, hypotf, hypotl, ldexp, log, log10, modf, pow,
sin, sinh, sqrt, tan, tanh

<**reg*.h**>   Special function register declarations for all supported
derivatives.

<**setjmp.h**>   longjmp, setjmp

• • • • • • • • •

<**stdarg.h**>   va_arg, va_end, va_start

<**signal.h**>   raise, signal. Functions are delivered as skeletons.

<**stddef.h**>   offsetof. Definition of special types.

<**stdio.h**>   clearerr, _close, fclose, feof, ferror, fflush, fgetc, fgetpos,
fgets, fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek,
fsetpos, ftell, fwrite, getc, getchar, gets, _lseek, _open, perror,
printf, putc, putchar, puts, _read, remove, rename, rewind,
scanf, setbuf, setvbuf, sprintf, sscanf, tmpfile, tmpnam,
ungetc, vfprintf, vprintf, vsprintf, _unlink, _write

<**stdlib.h**>   abort, abs, atexit, atof, atoi, atol, bsearch, calloc, div, exit,
fcalloc, ffree, fmalloc, frealloc, free, getenv, hcalloc, hfree,
hmalloc, hrealloc, labs, ldiv, malloc, mblen, mbstowcs,
mbtowc, ncalloc, nfree, nmalloc, nrealloc, qsort, rand,
realloc, scalloc, sfree, smalloc, srand, srealloc, strtod, strtol,
strtoul, wcstombs, wctomb

<**string.h**>   memchr, memcmp, memcpffb, memcpffw, memcpfhb,
memcpfhw, memcpfnb, memcpfnw, memcpfsb, memcpfsw,
memcphfb, memcphfw, memcphhb, memcphhw,
memcphnb, memcphnw, memcphsb, memcphsw, memcpnfb,
memcpnfw, memcpnhb, memcpnhw, memcpnnb,
memcpnnw, memcpnsb, memcpnsw, memcpsfb, memcpsfw,
memcpshb, memcpshw, memcpsnb, memcpsnw, memcpssb,
memcpssw, memcpy, memmove, memset, strcat, _fstrcat,
_hstrcat, _sstrcat, strchr, _fstrchr, _hstrchr, _sstrchr, strcmp,
_fstrcmp, _hstrcmp, _sstrcmp, strcol, strcpy, _fstrcpy,
_hstrcpy, _sstrcpy, strcspn, _fstrcspn, _hstrcspn, _sstrcspn,
strerror, strlen, _fstrlen, _hstrlen, _sstrlen, strncat, _fstrncat,
_hstrncat, _sstrncat, strncmp, _fstrncmp, _hstrncmp,
_sstrncmp, strncpy, _fstrncpy, _hstrncpy, _sstrncpy, strpbrk,
_fstrpbrk, _hstrpbrk, _sstrpbrk, strrchr, _fstrrchr, _hstrrchr,
_sstrrchr, strspn, _fstrspn, _hstrspn, _sstrspn, strstr, _fstrstr,
_hstrstr, _sstrstr, strtok, _fstrtok, _hstrtok, _sstrtok, strxfrm

<**time.h**>   asctime, clock, ctime, difftime, gmtime, localtime, mktime,
_stime, strftime, time, _tzset

<**unistd.h**>   Non-ANSI C header file with prototypes for standard POSIX
I/O functions. access, chdir, close, getcwd, lseek, read, stat,
lstat, fstat, unlink, write.

\<vt100.h\>   VT100 Terminal Emulation escape sequences for use with the
                CrossView Pro FSS feature.

\<wchar.h\>

fwprintf, wprintf, swprintf, vfwprintf, vwprintf, vswprintf,
fwscanf, wscanf, swscanf, fgetwc, fgetws, fputwc, fputws,
fwide, getwc, getwchar, putwc, putwchar, ungetwc, wcstod,
wcstol, wcstoul, wcscpy, wcsncpy, wcscat, wcsncat, wcscmp,
wcscoll, wcsncmp, wcsxfrm, wcschr, wcscspn, wcspbrk,
wcsrchr, wcsspn, wcsstr, wcstok, wcslen, wmemchr,
wmemcmp, wmemcpy, wmemmove, wmemset, wcsftime,
btowc, wctob, mbsinit, mbrlen, mbrtowc, wcrtomb,
mbsrtowcs, wcsrtombs

\<wctype.h\>

iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower,
iswprint, iswpunct, iswspace, iswupper, iswxdigit, towlower,
towupper, wctype, iswctype, wctrans, towctrans

6.6   C LIBRARY INTERFACE DESCRIPTION

Library functions that take `void` pointers as parameters imply default
memory. So, in that case explicit memory qualifiers, such as `_huge`, are
not allowed.

_close

```
#include <stdio.h>
int _close( int fd );
```

Low level file close function. _close is used by the functions close and
fclose. The given file descriptor should be properly closed, any buffer is
already flushed.

_fstrcat

```
#include <string.h>
char far *_fstrcat( char far *s, const char far *ct );
```

Concatenates far string `ct` to far string `s`, including the trailing NULL
character.

Returns   s

_fstrchr

```
#include <string.h>
char far *_fstrchr( const char far *cs, int c );
```

Returns   a far pointer to the first occurrence of character `c` in the
string `cs`. If not found, NULL is returned.

LIBRARIES

_fstrcmp

```
#include <string.h>
int _fstrcmp( const char far *cs,
              const char far *ct );
```

Compares far string `cs` to far string `ct`.

Returns   <0 if `cs` < `ct`,
            0   if `cs` == `ct`,
            >0 if `cs` > `ct`.

_fstrcpy

```
#include <string.h>
char far *_fstrcpy( char far *s, const char far *ct );
```

Copies far string `ct` into the far string `s`, including the trailing NULL
character.

Returns   s

_fstrcspn

```
#include <string.h>
size_t _fstrcspn( const char far *cs,
                  const char far *ct );
```

Returns   the length of the prefix in far string `cs`, consisting of
            characters not in the far string `ct`.

_fstrlen

```
#include <string.h>
size_t _fstrlen( const char far *cs );
```

Returns   the length of the far string in `cs`, not counting the NULL
            character.

_fstrncat

```
#include <string.h>
char far *_fstrncat( char far *s,
                     const char far *ct,
                     size_t n );
```

Concatenates far string `ct` to far string `s`, at most `n` characters are copied.
Add a trailing NULL character.

Returns   s

_fstrncmp

```
#include <string.h>
int _fstrncmp( const char far *cs,
               const char far *ct,
               size_t n );
```

Compares at most `n` bytes of far string `cs` to far string `ct`.

Returns   <0 if `cs` < `ct`,
            0   if `cs` == `ct`,
            >0 if `cs` > `ct`.

_fstrncpy

```
#include <string.h>
char far *_fstrncpy( char far *s,
                     const char far *ct,
                     size_t n );
```

Copies far string `ct` onto the far string `s`, at most `n` characters are copied.
Add a trailing NULL character if the string is smaller than `n` characters.

Returns   s

LIBRARIES

_fstrpbrk

```
#include <string.h>
char far *_fstrpbrk( const char far *cs,
                     const char far *ct );
```

Returns   a far pointer to the first occurrence in `cs` of any character out
of far string `ct`. If none are found, NULL is returned.

_fstrrchr

```
#include <string.h>
char far *_fstrrchr( const char far *cs, int c );
```

Returns   a far pointer to the last occurrence of `c` in the far string `cs`. If
not found, NULL is returned.

_fstrspn

```
#include <string.h>
size_t _fstrspn( const char far *cs,
                 const char far *ct );
```

Returns   the length of the prefix in far string `cs`, consisting of
characters in the far string `ct`.

_fstrstr

```
#include <string.h>
char far *_fstrstr( const char far *cs,
                    const char far *ct );
```

Returns   a far pointer to the first occurrence of far string `ct` in the far
string `cs`. Returns NULL if not found.

_fstrtok

```
#include <string.h>
char far *_fstrtok( char far *s, const char far *ct );
```

Search the far string **s** for tokens delimited by characters from far 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.

_hstrcat

```
#include <string.h>
char huge *_hstrcat( char huge *s,
                     const char huge *ct );
```

Concatenates huge string **ct** to huge string **s**, including the trailing NULL
character.

Returns   s

_hstrchr

```
#include <string.h>
char huge *_hstrchr( const char huge *cs, int c );
```

Returns   a huge pointer to the first occurrence of character **c** in the
                 string **cs**. If not found, NULL is returned.

_hstrcmp

```
#include <string.h>
int _hstrcmp( const char huge *cs,
              const char huge *ct );
```

Compares huge string **cs** to huge string **ct**.

Returns   <0 if **cs < ct**,
                 0   if **cs == ct**,
                 >0 if **cs > ct**.

LIBRARIES

_hstrcpy

```
#include <string.h>
char huge *_hstrcpy( char huge *s,
                     const char huge *ct );
```

Copies huge string `ct` into the huge string `s`, including the trailing NULL
character.

Returns   s

_hstrcspn

```
#include <string.h>
size_t _hstrcspn( const char huge *cs,
                  const char huge *ct );
```

Returns   the length of the prefix in huge string `cs`, consisting of
                characters not in the huge string `ct`.

_hstrlen

```
#include <string.h>
size_t _hstrlen( const char huge *cs );
```

Returns   the length of the huge string in `cs`, not counting the NULL
                character.

_hstrncat

```
#include <string.h>
char huge *_hstrncat( char huge *s,
                      const char huge *ct,
                      size_t n );
```

Concatenates huge string `ct` to huge string `s`, at most `n` characters are
copied. Add a trailing NULL character.

Returns   s

_hstrncmp

```
#include <string.h>
int _hstrncmp( const char huge *cs,
               const char huge *ct,
               size_t n );
```

Compares at most **n** bytes of huge string **cs** to huge string **ct**.

Returns   <0 if **cs** < **ct**,
                 0   if **cs** == **ct**,
                 >0 if **cs** > **ct**.

_hstrncpy

```
#include <string.h>
char huge *_hstrncpy( char huge *s,
                      const char huge *ct,
                      size_t n );
```

Copies huge string **ct** onto the huge string **s**, at most **n** characters are copied. Add a trailing NULL character if the string is smaller than **n** characters.

Returns   s

_hstrpbrk

```
#include <string.h>
char huge *_hstrpbrk( const char huge *cs,
                      const char huge *ct );
```

Returns   a huge pointer to the first occurrence in **cs** of any character out of huge string **ct**. If none are found, NULL is returned.

_hstrrchr

```
#include <string.h>
char huge *_hstrrchr( const char huge *cs, int c );
```

Returns   a huge pointer to the last occurrence of **c** in the huge string **cs**. If not found, NULL is returned.

LIBRARIES

_hstrspn

```
#include <string.h>
size_t _hstrspn( const char huge *cs,
                 const char huge *ct );
```

Returns   the length of the prefix in huge string `cs`, consisting of
                characters in the huge string `ct`.

_hstrstr

```
#include <string.h>
char huge *_hstrstr( const char huge *cs,
                     const char huge *ct );
```

Returns   a huge pointer to the first occurrence of huge string `ct` in the
                huge string `cs`. Returns NULL if not found.

_hstrtok

```
#include <string.h>
char huge *_hstrtok( char huge *s,
                     const char huge *ct );
```

Search the huge string `s` for tokens delimited by characters from huge
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.

_lseek

```
#include <stdio.h>
off_t _lseek( int fd, off_t offset, int whence );
```

Low level file positioning function. _lseek is used by all file positioning
functions (fgetpos, fseek, fsetpos, ftell, rewind).

_open

```
#include <stdio.h>
int _open( const char *name, int flags );
```

Low level file open function. _open is used by the functions fopen and
freopen. The given file should be properly opened.

_read

```
#include <stdio.h>
size_t
_read( int fd, char *buffer, size_t size );
```

Low level block input function. It reads a block of characters from the
given stream. This function interfaces to CrossView Pro's I/O Simulation
feature.

Returns   the number of characters read.

_stime

```
#include <time.h>
void _stime( time_t *s );
```

Sets the current calendar time.

Returns   nothing.

_sstrcat

```
#include <string.h>
char shuge *_sstrcat( char shuge *s,
                      const char shuge *ct );
```

Concatenates shuge string **ct** to shuge string **s**, including the trailing NULL
character.

Returns   s

_sstrchr

```
#include <string.h>
char shuge *_sstrchr( const char shuge *cs, int c );
```

Returns   a shuge pointer to the first occurrence of character **c** in the string **cs**. If not found, NULL is returned.

_sstrcmp

```
#include <string.h>
int _sstrcmp( const char shuge *cs,
              const char shuge *ct );
```

Compares shuge string **cs** to shuge string **ct**.

Returns   <0 if **cs < ct**,
                0 if **cs == ct**,
                >0 if **cs > ct**.

_sstrcpy

```
#include <string.h>
char shuge *_sstrcpy( char shuge *s,
                      const char shuge *ct );
```

Copies shuge string **ct** into the shuge string **s**, including the trailing NULL character.

Returns   s

_sstrcspn

```
#include <string.h>
size_t _sstrcspn( const char shuge *cs,
                  const char shuge *ct );
```

Returns   the length of the prefix in shuge string **cs**, consisting of characters not in the shuge string **ct**.

_sstrlen

```
#include <string.h>
size_t _sstrlen( const char shuge *cs );
```

Returns   the length of the shuge string in `cs`, not counting the NULL
                character.

_sstrncat

```
#include <string.h>
char shuge *_sstrncat( char shuge *s,
                       const char shuge *ct,
                       size_t n );
```

Concatenates shuge string `ct` to shuge string `s`, at most `n` characters are
copied. Add a trailing NULL character.

Returns   s

_sstrncmp

```
#include <string.h>
int _sstrncmp( const char shuge *cs,
               const char shuge *ct,
               size_t n );
```

Compares at most `n` bytes of shuge string `cs` to shuge string `ct`.

Returns   <0 if `cs < ct`,
                0   if `cs == ct`,
                >0 if `cs > ct`.

LIBRARIES

_sstrncpy

```
#include <string.h>
char shuge *_sstrncpy( char shuge *s,
                       const char shuge *ct,
                       size_t n );
```

Copies shuge string `ct` onto the shuge string `s`, at most `n` characters are copied. Add a trailing NULL character if the string is smaller than `n` characters.

Returns   s

_sstrpbrk

```
#include <string.h>
char shuge *_sstrpbrk( const char shuge *cs,
                       const char shuge *ct );
```

Returns   a shuge pointer to the first occurrence in `cs` of any character out of shuge string `ct`. If none are found, NULL is returned.

_sstrrchr

```
#include <string.h>
char shuge *_sstrrchr( const char shuge *cs, int c );
```

Returns   a shuge pointer to the last occurrence of `c` in the shuge string `cs`. If not found, NULL is returned.

_sstrspn

```
#include <string.h>
size_t _sstrspn( const char shuge *cs,
                 const char shuge *ct );
```

Returns   the length of the prefix in shuge string `cs`, consisting of characters in the shuge string `ct`.

. . . .

_sstrstr

```
#include <string.h>
char shuge *_sstrstr( const char shuge *cs,
                      const char shuge *ct );
```

Returns   a shuge pointer to the first occurrence of shuge string `ct` in
                the shuge string `cs`. Returns NULL if not found.

_sstrtok

```
#include <string.h>
char shuge *_sstrtok( char shuge *s,
                      const char shuge *ct );
```

Search the shuge string `s` for tokens delimited by characters from shuge
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.

_tolower

```
#include <ctype.h>
int _tolower( int c );
```

Converts `c` to a lowercase character, does not check if `c` really is an
uppercase character.

Returns   the converted character.

_toupper

```
#include <ctype.h>
int _toupper( int c );
```

Converts `c` to an uppercase character, does not check if `c` really is a
lowercase character.

Returns   the converted character.

_tzset

```
#include <time.h>
int _tzset( const char *s );
```

Converts the widely used time zone format string pointed to by **s** to tzone format. That string takes the form EST05EDT, where the number in the middle counts the hours West of UTC.

Returns   one if successful, or zero on error.

_unlink

```
#include <stdio.h>
int _unlink( const char *name );
```

Low level file remove function. _unlink is used by the function remove.

_write

```
#include <stdio.h>
size_t
_write( int fd, char *buffer, size_t count );
```

Low level block ouput function. It writes a block of characters to the given stream. This function interfaces to CrossView Pro's I/O Simulation feature.

Returns   the number of characters correctly written.

abort

```
#include <stdlib.h>
void abort( void );
```

Terminates the program abnormally.

Returns   nothing.

abs

```
#include <stdlib.h>
int abs( int n );
```

Returns   the absolute value of the signed int argument.

access

```
#include <unistd.h>
int access( const char * name, int mode );
```

Use the file system simulation feature of CrossView Pro to check the permissions of a file on the host. **mode** specifies the type of access and is a bit pattern constructed by a logical OR of the following values:

R_OK   Checks read permission.
W_OK   Checks write permission.
X_OK   Checks execute (search) permission.
F_OK   Checks to see if the file exists.

Returns   zero if successful,
              -1 on error.

acos

```
#include <math.h>
double acos( double x );
```

Returns   the arccosine $\cos^{-1}(x)$ of x in the range $[0, \pi]$,
              $x \in [-1, 1]$.

asctime

```
#include <time.h>
char *asctime( const struct tm *tp );
```

Converts the time in the structure ***tp** into a string of the form:

```
Mon Jan 21 16:15:14 1993\n\0
```

Returns   the time in string form.

LIBRARIES

asin

```
#include <math.h>
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>
assert( expr );
```

When compiled with NDEBUG, this is an empty macro. When compiled without NDEBUG defined, it checks if 'expr' is true or false. If it is false, then a line like:

```
"Assertion failed: expression, file filename, line num"
```

is printed.

Returns   nothing.

atan

```
#include <math.h>
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>
double atan2( double y, double x );
```

Returns   the result of: $\tan^{-1}(y/x)$ in the range $[-\pi, \pi]$.

atexit

```
#include <stdlib.h>
int atexit( void (*fcn)( void ) );
```

Registers the function fcn to be called when the program terminates normally.

Returns   zero, if program terminates normally.
            non-zero, if the registration cannot be made.

atof

```
#include <stdlib.h>
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>
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>
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.

LIBRARIES

bsearch

```
#include <stdlib.h>
void *bsearch( const void *key,
               const void *base, size_t n,
               size_t size, 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.

btowc

```
#include <wchar.h>
wint_t btowc( int c );
```

Determines whether **c** constitutes a valid single-byte character in the initial shift state.

Returns   WEOF if **c** has the value EOF or if `(unsigned char)c` does not constitute a valid single-byte character in the initial shift state. Otherwise, it returns the wide character representation of that character.

calloc

```
#include <stdlib.h>
void *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 7.3, *Heap Size*). By default no heap is allocated.

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>
double ceil( double x );
```

Returns   the smallest integer not less than **x**, as a double.

chdir

```
#include <unistd.h>
int chdir( const char *path );
```

Use the file system simulation feature of CrossView Pro to change the current directory on the host to the directory indicated by **path**.

Returns   zero if successful,
                -1 on error.

clearerr

```
#include <stdio.h>
void clearerr( FILE *stream );
```

Clears the end of file and error indicators for **stream**.

Returns   nothing.

clock

```
#include <time.h>
clock_t clock( void );
```

To perform real-time clock support, you must customize this function. See the file **time.c** in the **examples\time** directory demonstrating an implementation of this low-level time function.

Returns   the processor time used. To determine the time used in seconds, the value returned must be divided by the value of the macro **CLOCKS_PER_SEC**, as defined in **time.h** If the processor time used is not available or its value cannot be represented, the function returns the value **(clock_t)-1**.

LIBRARIES

close

```
#include <unistd.h>
int close( int fd );
```

File close function. The given file descriptor should be properly closed. This function calls _close.

Returns   zero if successful,
-1 on error.

cos

```
#include <math.h>
double cos( double x );
```

Returns   the cosine of x.

cosh

```
#include <math.h>
double cosh( double x );
```

Returns   the hyperbolic cosine of x.

ctime

```
#include <time.h>
char *ctime( const time_t *tp );
```

Converts the calender time *tp into local time, in string form. This function is the same as:

```
asctime( localtime( tp ) );
```

Returns   the local time in string form.

difftime

```
#include <time.h>
double difftime( time_t time2, time_t time1 );
```

Computes the difference between calendar times.

Returns   the result of `time2 - time1` in seconds.

div

```
#include <stdlib.h>
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>
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>
double exp( double x );
```

Returns   the result of the exponential function e^x.

fabs

```
#include <math.h>
double fabs( double x );
```

Returns   the absolute double value of **x**. $|x|$

fcalloc

```
#include <stdlib.h>
void _far *fcalloc( size_t nobj, size_t size );
```

Far variant of "calloc()". See section 7.3, *Heap Size*.

fclose

```
#include <stdio.h>
int fclose( FILE *stream )
```

Flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, then closes the **stream**.

Returns   zero if the **stream** is successfully closed, or EOF on error.

feof

```
#include <stdio.h>
int feof( FILE *stream );
```

Returns   a non-zero value if the end-of-file indicator for **stream** is set.

ferror

```
#include <stdio.h>
int ferror( FILE *stream );
```

Returns   a non-zero value if the error indicator for **stream** is set.

fflush

```
#include <stdio.h>
int fflush( FILE *stream );
```

Writes any buffered but unwritten date, if `stream` is an output stream. If `stream` is an input stream, the effect is undefined.

Returns   zero if successful, or EOF on a write error.

ffree

```
#include <stdlib.h>
void ffree( void _far *p );
```

Deallocates the space pointed to by `p`. `p` Must point to space earlier allocated by a call to "fcalloc()", "fmalloc()" or "frealloc()". Otherwise the behavior is undefined.

Returns   nothing

fgetc

```
#include <stdio.h>
int fgetc( FILE *stream );
```

Reads one character from the given `stream`.

Returns   the read character, or EOF on error.

fgetpos

```
#include <stdio.h>
int fgetpos( FILE *stream, fpos_t *ptr );
```

Stores the current value of the file position indicator for the stream pointed to by `stream` in the object pointed to by `ptr`. The type `fpos_t` is suitable for recording such values.

Returns   zero if successful,
                  a non-zero value on error.

fgets

```
#include <stdio.h>
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.

fgetwc

```
#include <wchar.h>
wint_t fgetwc( FILE *stream );
```

Reads one wide character from the given `stream`.

Returns   the read wide character, or WEOF on error.

fgetws

```
#include <wchar.h>
wchar_t *fgetws( wchar_t *s, int n, FILE *stream );
```

Reads at most the next n-1 wide characters from the given `stream` into the array `s` until a newline is found.

Returns   s, or NULL on end-of-file or error.

floor

```
#include <math.h>
double floor( double x );
```

Returns   the largest integer not greater than `x`, as a double.

fmalloc

```
#include <stdlib.h>
void _far *fmalloc( size_t size );
```

Far variant of "malloc()". See section 7.3, *Heap Size*.

fmod

```
#include <math.h>
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.

fopen

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
```

Opens a file for a given mode.

Returns   a stream. If the file cannot not be opened, NULL is returned.

fopen needs a heap size of at least 512 bytes.

You can specify the following values for mode:

"r"   read; open text file for reading

"w"   write; create text file for writing; if the file already exists its contents is discarded

"a"   append; open existing text file or create new text file for writing at end of file

"r+"   open text file for update; reading and writing

"w+"   create text file for update; previous contents if any is discarded

"a+"   append; open or create text file for update, writes at end of file

The update mode (with a '+') allows reading and writing of the same file. In this mode the function fflush must be called between a read and a write or vice versa. By including the letter "b" after the initial letter, you can indicate that the file is a binary file. E.g. "rb" means read binary, "w+b" means create binary file for update. The filename is limited to FILENAME_MAX characters. At most FOPEN_MAX files may be open at once.

fprintf

```
#include <stdio.h>
int fprintf( FILE *stream, const char *format, ... );
```

Performs a formatted write to the given `stream`.

See also "printf()" and "_write()".

fputc

```
#include <stdio.h>
int fputc( int c, FILE *stream );
```

Puts one character onto the given `stream`.

See also "_write()".

Returns   EOF on error.

fputs

```
#include <stdio.h>
int fputs( const char *s, FILE *stream );
```

Writes the string to a `stream`. The terminating NULL character is not written.

See also "_write()".

Returns   0 if successful, or EOF on error.

fputwc

```
#include <wchar.h>
wint_t fputwc( int c, FILE *stream );
```

Puts one wide character onto the given `stream`.

Returns   the wide character written or WEOF on error.

fputws

```
#include <wchar.h>
int fputws( const wchar_t *s, FILE *stream );
```

Writes the wide string to a `stream`. The terminating NULL wide character
is not written.

Returns   0 if successful, or EOF on error.

fread

```
#include <stdio.h>
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 "_read()".

Returns   the number of successfully read objects.

frealloc

```
#include <stdlib.h>
void _far *frealloc( void _far *p, size_t size );
```

Far variant of "realloc()". See section 7.3, *Heap Size*.

free

```
#include <stdlib.h>
void free( void *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

LIBRARIES

freopen

```
#include <stdio.h>
FILE *freopen( const char *filename,
               const char *mode, FILE *stream );
```

Opens a file for a given `mode` associates the `stream` with it. This function is normally used to change the files associated with stdin, stdout, or stderr.

See also "fopen()".

Returns   `stream`, or NULL on error.

frexp

```
#include <math.h>
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>
int fscanf( FILE *stream, const char *format, ... );
```

Performs a formatted read from the given `stream`.

See also "scanf()" and "_read()".

Returns   the number of items converted successfully.

fseek

```
#include <stdio.h>
int fseek( FILE *stream, long offset, int origin );
```

Sets the file position indicator for `stream`. A subsequent read or write will access data beginning at the new position. For a binary file, the position is set to `offset` characters from `origin`, which may be SEEK_SET for the beginning of the file, SEEK_CUR for the current position in the file, or SEEK_END for the end-of-file. For a text stream, `offset` must be zero, or a value returned by `ftell`. In this case `origin` must be SEEK_SET.

Returns   zero if successful,
              a non-zero value on error.

fsetpos

```
#include <stdio.h>
int fsetpos( FILE *stream, const fpos_t *ptr );
```

Positions `stream` at the position recorded by `fgetpos` in `*ptr`.

Returns   zero if successful,
              a non-zero value on error.

fstat

```
#include <unistd.h>
int fstat( int fd, struct stat * buf );
```

This function is identical to `stat()`, except that it uses a file descriptor instead of a name.

Returns   zero if successful,
              -1 on error.

ftell

```
#include <stdio.h>
long ftell( FILE *stream );
```

Returns   the current file position for `stream`, or
               `-1L` on error.

fwide

```
#include <wchar.h>
int fwide( FILE *stream, int mode );
```

Determines the orientation of the `stream`. If `mode` is greater than zero, the
function first attempts to make the stream wide oriented. If `mode` is less
than zero, the function first attempts to make the stream byte oriented.
Otherwise, `mode` is zero and the function does not alter the orientation of
the stream.

Returns   a value greater than zero if, after the call, the stream has wide
               orientation, a value less than zero if the stream has byte
               orientation, or zero if the stream has no orientation.

fwprintf

```
#include <wchar.h>
int fwprintf( FILE *stream,
              const wchar_t *format, ... );
```

Writes output to the given `stream` under control of the wide string
pointed to by `format` that specifies how subsequent arguments are
converted for output.

See also "printf()".

Returns   the number of wide characters transmitted, or a negative
               value if an output or encoding error occurred.

fwrite

```
#include <stdio.h>
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.

fwscanf

```
#include <wchar.h>
int fwscanf( FILE *stream,
             const wchar_t *format, ... );
```

Reads input from the given `stream`, under control of the wide string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

See also "scanf()".

Returns   the number of input items assigned or EOF on error.

getc

```
#include <stdio.h>
int getc( FILE *stream );
```

Reads one character out of the given `stream`.

See also "_read()".

Returns   the character read or EOF on error.

getchar

```
#include <stdio.h>
int getchar( void );
```

Reads one character from standard input.

See also "_read()".

Returns   the character read or EOF on error.

getcwd

```
#include <unistd.h>
char * getcwd( char * buf, size_t size );
```

Use the file system simulation feature of CrossView Pro to retrieve the current directory on the host.

Returns   the directory name if successful,
            NULL on error.

getenv

```
#include <stdlib.h>
char *getenv( const char *name );
```

Returns   the environment string associated with name, or NULL if no
            string exists.

gets

```
#include <stdio.h>
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 "_read()".

Returns   a pointer to the read string or NULL on error.

getwc

```
#include <wchar.h>
wint_t getwc( FILE *stream );
```

Reads one wide character out of the given `stream`.

Returns   the wide character read, or WEOF on error.

getwchar

```
#include <wchar.h>
wint_t getwchar( void );
```

Reads one wide character from standard input.

Returns   the wide character read, or WEOF on error.

gmtime

```
#include <time.h>
struct tm *gmtime( const time_t *tp );
```

Converts the calender time `*tp` into Coordinated Universal Time (UTC).

Returns   a structure representing the UTC, or NULL if UTC is not
available.

hcalloc

```
#include <stdlib.h>
void _huge *hcalloc( unsigned long nobj,
                     unsigned long size );
```

Huge variant of "calloc()". See section 7.3, *Heap Size*.

LIBRARIES

hfree

```
#include <stdlib.h>
void hfree( void _huge *p );
```

Deallocates the space pointed to by **p**. **p** Must point to space earlier allocated by a call to "hcalloc()", "hmalloc()" or "hrealloc()". Otherwise the behavior is undefined.

Returns   nothing

hmalloc

```
#include <stdlib.h>
void _huge *hmalloc( unsigned long size );
```

Huge variant of "malloc()". See section 7.3, *Heap Size*.

hrealloc

```
#include <stdlib.h>
void _huge *hrealloc( void _huge *p,
                      unsigned long size );
```

Huge variant of "realloc()". See section 7.3, *Heap Size*.

hypot

```
#include <math.h>
double hypot( double x, double y );
```

Returns   the hypotenuse for the given values, as a double.

hypotf

```
#include <math.h>
float hypotf( float x, float y );
```

Returns   the hypotenuse for the given values, as a float.

hypotl

```
#include <math.h>
long double hypotl( long double x, long double y );
```

Returns   the hypotenuse for the given values, as a long double.

isalnum

```
#include <ctype.h>
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>
int isalpha( int c );
```

Returns   a non-zero value when **c** is an alphabetic character
                ([A-Z][a-z]).

isascii

```
#include <ctype.h>
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>
int iscntrl( int c );
```

Returns   a non-zero value when **c** is a control character.

LIBRARIES

isdigit

```
#include <ctype.h>
int isdigit( int c );
```

Returns   a non-zero value when **c** is a numeric character ([0-9]).

isgraph

```
#include <ctype.h>
int isgraph( int c );
```

Returns   a non-zero value when **c** is printable, but not a space.

isinf

```
#include <float.h>
int isinf( double d );
```

IEEE-754-1985 recommended function. Test the given variable on being an infinite (IEEE-754) value.

Returns   zero if the variable is not +-infinite, else non-zero.

isinff

```
#include <float.h>
int isinff( float f );
```

IEEE-754-1985 Recommended function. Test the given variable on being an infinite (IEEE-754) value.

Returns   zero if the variable is not +-infinite, else non-zero.

islower

```
#include <ctype.h>
int islower( int c );
```

Returns   a non-zero value when **c** is a lowercase character ([a-z]).

isnan

```
#include <float.h>
int isnan( double d );
```

IEEE-754-1985 recommended function. Test the given variable on being a NaN (Not a Number, IEEE-754) value.

Returns   zero if the variable is not NaN, else non-zero.

isnanf

```
#include <float.h>
int isnanf( float f );
```

IEEE-754-1985 Recommended function. Test the given variable on being a NaN (Not a Number, IEEE-754) value.

Returns   zero if the variable is not NaN, else non-zero.

isprint

```
#include <ctype.h>
int isprint( int c );
```

Returns   a non-zero value when c is printable, including spaces.

ispunct

```
#include <ctype.h>
int ispunct( int c );
```

Returns   a non-zero value when c is a punctuation character (such as
                  '.', ',', '!', etc.).

isspace

```
#include <ctype.h>
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>
int isupper( wint_t wc );
```

Returns   a non-zero value when **c** is an uppercase character ([A-Z]).

iswalnum

```
#include <wctype.h>
int iswalnum( wint_t wc );
```

Returns   a non-zero value when **wc** is an alphabetic wide character or
            a number ([A-Z][a-z][0-9]).

iswalpha

```
#include <wctype.h>
int iswalpha( wint_t wc );
```

Returns   a non-zero value when **wc** is an alphabetic wide character
            ([A-Z][a-z]).

iswcntrl

```
#include <wctype.h>
int iswcntrl( wint_t wc );
```

Returns   a non-zero value when **wc** is a control wide character.

iswctype

```
#include <wctype.h>
int iswctype( wint_t wc, wctype_t desc );
```

Returns   a non-zero value (true) if and only if the value of the wide
            character **wc** has the property described by **desc**.

For example, the function `iswalnum(wc)` is the same as specifying:

```
iswctype(wc, wctype("alnum"))
```

iswdigit

```
#include <wctype.h>
int iswdigit( wint_t wc );
```

Returns   a non-zero value when **wc** is a numeric character ([0-9]).

iswgraph

```
#include <wctype.h>
int iswgraph( wint_t wc );
```

Returns   a non-zero value when **wc** is printable, but not a space.

iswlower

```
#include <wctype.h>
int iswlower( wint_t wc );
```

Returns   a non-zero value when **wc** is a lowercase wide character
               ([a-z]).

iswprint

```
#include <wctype.h>
int iswprint( wint_t wc );
```

Returns   a non-zero value when **wc** is printable, including spaces.

iswpunct

```
#include <wctype.h>
int iswpunct( wint_t wc );
```

Returns   a non-zero value when **wc** is a punctuation wide character
               (such as '.', ',', '!', etc.).

iswspace

```
#include <wctype.h>
int iswspace( wint_t wc );
```

Returns   a non-zero value when **wc** is a white-space wide character
                (space, tab, vertical tab, formfeed, linefeed, carriage return).

iswupper

```
#include <wctype.h>
int iswupper( wint_t wc );
```

Returns   a non-zero value when **wc** is an uppercase wide character
                ([A-Z]).

iswxdigit

```
#include <wctype.h>
int iswxdigit( wint_t wc );
```

Returns   a non-zero value when **wc** is a hexadecimal digit
                ([0-9][A-F][a-f]).

isxdigit

```
#include <ctype.h>
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>
long labs( long n );
```

Returns   the absolute value of the signed long argument.

ldexp

```
#include <math.h>
double ldexp( double x, int n );
```

Returns   the result of: $x \cdot 2^n$.

ldiv

```
#include <stdlib.h>
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`.

localeconv

```
#include <locale.h>
struct lconv *localeconv( void );
```

Sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale.

Returns   a pointer to the filled-in object.

localtime

```
#include <time.h>
struct tm *localtime( const time_t *tp );
```

Converts the calender time `*tp` into local time.

Returns   a structure representing the local time.

log

```
#include <math.h>
double log( double x );
```

Returns   the natural logarithm `ln(x), x>0`.

log10

```
#include <math.h>
double log10( double x );
```

Returns   the base 10 logarithm `log10(x), x>0`.

longjmp

```
#include <setjmp.h>
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` may not be zero.

Returns   nothing.

lstat

```
#include <unistd.h>
int lstat( const char * name, struct stat * buf );
```

This function is identical to `stat()`, except in the case of a symbolic link, where the link itself is 'stat'-ed, not the file that it refers to.

Returns   zero if successful,
            -1 on error.

malloc

```
#include <stdlib.h>
void *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 7.3,
Heap Size). By default no heap is allocated.

Returns   a pointer to space in external memory of `size` bytes length.
NULL if there is not enough space left.

mblen

```
#include <stdlib.h>
int mblen( const char *s, size_t n );
```

Determines the number of bytes comprising the multi-byte character
pointed to by `s`, if `s` is not a null pointer. Except that the shift state is not
affected. At most `n` characters will be examined, starting at the character
pointed to by `s`.

Returns   the number of bytes, or 0 if `s` points to the null character, or
-1 if the bytes do not form a valid multi-byte character.

mbrlen

```
#include <wchar.h>
size_t mbrlen( const char *s, size_t n,
               mbstate_t *ps);
```

Is equivalent to the call:

```
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal)
```

where `internal` is the `mbstate_t` object for the `mbrlen` function,
except that the expression designated by `ps` is evaluated only once.

Returns   a value between zero and `n`, inclusive, `(size_t)(-2)`, or
`(size_t)(-1)`.

LIBRARIES

mbrtowc

```
#include <wchar.h>
size_t mbrtowc( wchar_t *pwc, const char *s,
                size_t n, mbstate_t *ps );
```

Inspects at most **n** bytes beginning with the byte pointed to by **s** to
determine the number of bytes needed to complete the next multi-byte
character (including any shift sequences). If the function determines that
the next multi-byte character is complete and valid, it determines the
value of the corresponding wide character and then, if **pwc** is not a NULL
pointer, stores that value in the object pointed to by **pwc**. If the
corresponding wide character is the NULL wide character, the resulting
state described is the initial conversion state.

Returns   the number of bytes, or 0 if **s** points to the null character, or
               (size_t)(-2) if the bytes form an incomplete (but
               potentionally valid) multi-byte character, or **(size_t)(-1)**
               if the bytes do not form a valid multi-byte character.

mbsinit

```
#include <wchar.h>
int mbsinit( const mbstate_t *ps );
```

Determines whether the pointed-to **mbstate_t** object describes an initial
conversion state, if **ps** is not a NUL pointer.

Returns   non-zero if **ps** is a NULL pointer or if the pointed-to object
               describes an initial conversion state. Otherwise, it returns
               zero.

mbsrtowcs

```
#include <wchar.h>
size_t mbsrtowcs( wchar_t *dst, const char **src,
                  size_t len, mbstate_t *ps );
```

Converts a sequence of multi-byte characters that begins in the conversion
state described by the object pointed to by `ps`, from the array indirectly
pointed to by `src` into a sequence of corresponding wide characters. This
function then stores the converted characters into the array pointed to by
`dst`, stopping when `len` wide characters have been stored, or when a
sequence of bytes is encountered that does not form a valid multi-byte
character, or if a null wide character is stored.

Returns   the number of multi-byte characters successfully converted
               (not including the terminating null character, if any), or
               (`size_t`)-1 if an invalid multi-byte character is
               encountered.

mbstowcs

```
#include <stdlib.h>
size_t mbstowcs( wchar_t *pwcs,
                 const char *s, size_t n );
```

Converts a sequence of multi-byte characters that begins in the initial shift
state from the array pointed to by `s`, into a sequence of corresponding
wide characters and stores these wide characters into the array pointed to
by `pwcs`, stopping after `n` wide characters are stored or a null wide
character is stored.

Returns   the number of array elements modified (not including a
               terminating null wide character, if any), or (`size_t`)-1 if an
               invalid multi-byte character is encountered.

LIBRARIES

mbtowc

```
#include <stdlib.h>
int mbtowc( wchar_t *pwc,
            const char *s, size_t n );
```

Determines the number of bytes that comprise the multi-byte character pointed to by **s**. It then determines the value of the wide character that corresponds to that multi-byte character. If the multi-byte character is valid and **pwc** is not a null pointer, the **mbtowc** function stores the value of the wide character in the object pointed to by **pwc**. At most **n** bytes will be examined, starting at the byte pointed to by **s**.

Returns   the number of bytes, or 0 if **s** points to the null wide character, or -1 if the bytes do not form a valid multi-byte character.

memchr

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

memcpffb

```
#include <string.h>
void memcpffb( void far *dest,
               void far *src, size_t n);
```

Copies n bytes from far data pointed by `src` to far data pointed by `dest`.
No care is taken if the two objects overlap and page boundaries are not
checked. (0 < n <= 16384)

Returns   nothing

memcpffw

```
#include <string.h>
void memcpffw( void far *dest,
               void far *src, size_t n);
```

Copies n words from far data pointed by `src` to far data pointed by `dest`.
No care is taken if the two objects overlap and page boundaries are not
checked. (0 < n <= 8192)

Returns   nothing

memcpfhb

```
#include <string.h>
void memcpfhb( void huge *dest,
               void far *src, size_t n);
```

Copies n bytes from far data pointed by `src` to huge data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for far data.
(0 < n <= 16384)

Returns   nothing

memcpfhw

```
#include <string.h>
void memcpfhw( void huge *dest,
               void far *src, size_t n);
```

Copies n words from far data pointed by `src` to huge data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for far data.
(0 < n <= 8192)

Returns   nothing

memcpfnb

```
#include <string.h>
void memcpfnb( void near *dest,
               void far *src, size_t n);
```

Copies n bytes from far data pointed by `src` to near data pointed by
`dest`. No care is taken if the two objects overlap and page boundaries are
not checked. (0 < n <= 16384)

Returns   nothing

memcpfnw

```
#include <string.h>
void memcpfnw( void near *dest,
               void far *src, size_t n);
```

Copies n words from far data pointed by `src` to near data pointed by
`dest`. No care is taken if the two objects overlap and page boundaries are
not checked. (0 < n <= 8192)

Returns   nothing

memcpfsb

```
#include <string.h>
void memcpfsb( void shuge *dest,
               void far *src, size_t n);
```

Copies n bytes from far data pointed by `src` to shuge data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for far data.
(0 < n <= 16384)

Returns   nothing

memcpfsw

```
#include <string.h>
void memcpfsw( void shuge *dest,
               void far *src, size_t n);
```

Copies n words from far data pointed by `src` to shuge data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for far data.
(0 < n <= 8192)

Returns   nothing

memcphfb

```
#include <string.h>
void memcphfb( void far *dest,
               void huge *src, size_t n);
```

Copies n bytes from huge data pointed by `src` to far data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for far data.
(0 < n <= 16384)

Returns   nothing

memcphfw

```
#include <string.h>
void memcphfw( void far *dest,
               void huge *src, size_t n);
```

Copies n words from huge data pointed by `src` to far data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for far data.
(0 < n <= 8192)

Returns   nothing

memcphhb

```
#include <string.h>
void memcphhb( void huge *dest,
               void huge *src, size_t n);
```

Copies n bytes from huge data pointed by `src` to huge data pointed by
`dest`. No care is taken if the two objects overlap.
(0 < n <= 65535)

Returns   nothing

memcphhw

```
#include <string.h>
void memcphhw( void huge *dest,
               void huge *src, size_t n);
```

Copies n words from huge data pointed by `src` to huge data pointed by
`dest`. No care is taken if the two objects overlap.
(0 < n <= 65535)

Returns   nothing

memcphnb

```
#include <string.h>
void memcphnb( void near *dest,
               void huge *src, size_t n);
```

Copies n bytes from huge data pointed by `src` to near data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for near data. (0 < n <= 16384)

Returns   nothing

memcphnw

```
#include <string.h>
void memcphnw( void near *dest,
               void huge *src, size_t n);
```

Copies n words from huge data pointed by `src` to near data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for near data. (0 < n <= 8192)

Returns   nothing

memcphsb

```
#include <string.h>
void memcphsb( void shuge *dest,
               void huge *src, size_t n);
```

Copies n bytes from huge data pointed by `src` to shuge data pointed by
`dest`. No care is taken if the two objects overlap.
(0 < n <= 65535)

Returns   nothing

memcphsw

```
#include <string.h>
void memcphsw( void shuge *dest,
               void huge *src, size_t n);
```

Copies n words from huge data pointed by `src` to shuge data pointed by
`dest`. No care is taken if the two objects overlap.
(0 < n <= 65535)

Returns   nothing

memcpnfb

```
#include <string.h>
void memcpnfb( void far *dest,
               void near *src, size_t n);
```

Copies n bytes from near data pointed by `src` to far data pointed by
`dest`. No care is taken if the two objects overlap.
(0 < n <= 16384)

Returns   nothing

memcpnfw

```
#include <string.h>
void memcpnfw( void far *dest,
               void near *src, size_t n);
```

Copies n words from near data pointed by `src` to far data pointed by
`dest`. No care is taken if the two objects overlap.
(0 < n <= 8192)

Returns   nothing

memcpnhb

```
#include <string.h>
void memcpnhb( void huge *dest,
               void near *src, size_t n);
```

Copies **n** bytes from near data pointed by `src` to huge data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for near data. (0 < **n** <= 16384)

Returns   nothing

memcpnhw

```
#include <string.h>
void memcpnhw( void huge *dest,
               void near *src, size_t n);
```

Copies **n** words from near data pointed by `src` to huge data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for huge data but not checked for near data. (0 < **n** <= 8192)

Returns   nothing

memcpnnb

```
#include <string.h>
void memcpnnb( void near *dest,
               void near *src, size_t n);
```

Copies **n** bytes from near data pointed by `src` to near data pointed by
`dest`. No care is taken if the two objects overlap and page boundaries are
not checked. (0 < **n** <= 16384)

Returns   nothing

LIBRARIES

memcpnnw

```
#include <string.h>
void memcpnnw( void near *dest,
               void near *src, size_t n);
```

Copies n words from near data pointed by **src** to near data pointed by
dest. No care is taken if the two objects overlap and page boundaries are
not checked. (0 < n <= 8192)

Returns   nothing

memcpnsb

```
#include <string.h>
void memcpnsb( void shuge *dest,
               void near *src, size_t n);
```

Copies n bytes from near data pointed by **src** to shuge data pointed by
dest. No care is taken if the two objects overlap. Page boundaries are
checked for shuge data but not for near data. (0 < n <= 16384)

Returns   nothing

memcpnsw

```
#include <string.h>
void memcpnsw( void shuge *dest,
               void near *src, size_t n);
```

Copies n words from near data pointed by **src** to shuge data pointed by
dest. No care is taken if the two objects overlap. Page boundaries are
checked for shuge data but not for near data. (0 < n <= 8192)

Returns   nothing

memcpsfb

```
#include <string.h>
void memcpsfb( void far *dest,
               void shuge *src, size_t n);
```

Copies n bytes from shuge data pointed by `src` to far data pointed by `dest`. No care is taken if the two objects overlap. Page boundaries are checked for shuge data but not checked for far data. (0 < n <= 16384)

Returns   nothing

memcpsfw

```
#include <string.h>
void memcpsfw( void far *dest,
               void shuge *src, size_t n);
```

Copies n words from shuge data pointed by `src` to far data pointed by `dest`. No care is taken if the two objects overlap. Page boundaries are checked for shuge data but not checked for far data. (0 < n <= 8192)

Returns   nothing

memcpshb

```
#include <string.h>
void memcpshb( void huge *dest,
               void shuge *src, size_t n );
```

Copies n bytes from shuge data pointed by `src` to huge data pointed by `dest`. No care is taken if the two objects overlap.
(0 < n <= 16384)

Returns   nothing

memcpshw

```
#include <string.h>
void memcpshw( void huge *dest,
               void shuge *src, size_t n);
```

Copies n words from shuge data pointed by `src` to huge data pointed by
`dest`. No care is taken if the two objects overlap.
(0 < n <= 8192)

Returns   nothing

memcpsnb

```
#include <string.h>
void memcpsnb( void near *dest,
               void shuge *src, size_t n);
```

Copies n bytes from shuge data pointed by `src` to near data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for shuge data but not for near data.
(0 < n <= 16384)

Returns   nothing

memcpsnw

```
#include <string.h>
void memcpsnw( void near *dest,
               void shuge *src, size_t n);
```

Copies n words from shuge data pointed by `src` to near data pointed by
`dest`. No care is taken if the two objects overlap. Page boundaries are
checked for shuge data but not for near data.
(0 < n <= 8192)

Returns   nothing

memcpssb

```
#include <string.h>
void memcpssb( void shuge *dest,
               void shuge *src, size_t n);
```

Copies **n** bytes from shuge data pointed by **src** to shuge data pointed by **dest**. No care is taken if the two objects overlap.
(0 < n <= 16384)

Returns   nothing

memcpssw

```
#include <string.h>
void memcpssw( void shuge *dest,
               void shuge *src, size_t n);
```

Copies **n** words from shuge data pointed by **src** to shuge data pointed by **dest**. No care is taken if the two objects overlap.
(0 < n <= 8192)

Returns   nothing

memcpy

```
#include <string.h>
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>
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

LIBRARIES

memset

```
#include <string.h>
void *memset( void *s, int c, size_t n );
```

Fills the first **n** bytes of **s** with character **c**.

Returns   **s**

mktime

```
#include <time.h>
time_t mktime( struct tm *tp );
```

Converts the local time in the structure ***tp** into calendar time.

Returns   the calendar time in seconds, or -1 if it cannot be
               represented.

modf

```
#include <math.h>
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.

ncalloc

```
#include <stdlib.h>
void _near *ncalloc( size_t nobj, size_t size );
```

Near variant of "calloc()". See section 7.3, *Heap Size*.

nfree

```
#include <stdlib.h>
void nfree( void _near *p );
```

Deallocates the space pointed to by **p**. **p** Must point to space earlier allocated by a call to "ncalloc()", "nmalloc()" or "nrealloc()". Otherwise the behavior is undefined.

Returns   nothing

nmalloc

```
#include <stdlib.h>
void _near *nmalloc( size_t size );
```

Near variant of "malloc()". See section 7.3, *Heap Size*.

nrealloc

```
#include <stdlib.h>
void _near *nrealloc( void _near *p, size_t size );
```

Near variant of "realloc()". See section 7.3, *Heap Size*.

offsetof

```
#include <stddef.h>
int offsetof( type, member );
```

Be aware, offsetof() for bit structures/unions may give unpredictable results. Also the offsetof() of a bitfield is undefined.

Returns   the offset for the given **member** in an object of type.

open

```
#include <fcntl.h>
int open( const char * name, int flags );
```

Opens a file a file for reading or writing. This function calls _open.

See also "fopen()".

Returns   the file descriptor if successful (a non-negative integer), or
               -1 on error.

perror

```
#include <stdio.h>
void perror( const char *s );
```

Prints **s** and an implementation-defined error message corresponding to
the integer **errno**, as if by:

  fprintf(stderr, "%s: %s\n", s, "*error message*");

The contents of the error message are the same as those returned by the
strerror function with the argument **errno**.

See also the "strerror()" function.

Returns   nothing.

pow

```
#include <math.h>
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>
int printf( const char *format, ... );
```

Performs a formatted write to the standard output stream.

See also "_write()".

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
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.

Character	Printed as
p	pointer; printed as a hexadecimal number, prefixed with: \<near\>, \<far\> or \<huge\>.
	For the different pointer types the following formats are used:
	\<near\> OOOO \<far\> PPPP:OOOO \<huge\> SS:OOOO
	where:
	O is offset P is page S is segment
%	No argument is converted, a '%' is printed.

Table 6-1: Printf conversion characters

The 'p' conversion character can be used to print pointers. In the tiny and small memory models, pointers will be printed as near pointers by default. In the medium and large memory models, pointers will be printed as far pointers by default. By specifying one of the length modifiers 'h', 'l' or 'L', a pointer will always be printed as near, far or huge respectively.

Because of the large overhead of the printf() function on small programs, three different versions of the formatter (`_doprint.c`) are delivered. The LARGE version is able to print everything as specified above. The MEDIUM version has no floating point formatting. When a floating point conversion character is found, `errno` is filled with the correct error number, printf stops immediately. The SMALL version does not print floating point, and does not accept flags, width specifier, period and precision. This formatter is considerably smaller in code size than the MEDIUM or LARGE version.

putc

```
#include <stdio.h>
int putc( int c, FILE *stream );
```

Puts one character onto the given stream.

See also "_write()".

Returns   EOF on error.

putchar

```
#include <stdio.h>
int putchar( int c );
```

Puts one character onto standard output.

See also "_write()".

Returns   the character written or EOF on error.

puts

```
#include <stdio.h>
int puts( const char *s );
```

Writes the string to stdout, the string is terminated by a newline.

See also "_write()".

Returns   NULL if successful, or EOF on error.

putwc

```
#include <wchar.h>
wint_t putwc( wchar_t c, FILE *stream );
```

Puts one wide character onto the given stream.

Returns   the wide character written, or WEOF on error.

putwchar

```
#include <wchar.h>
wint_t putwchar( wchar_t c );
```

Puts one wide character onto standard output.

Returns   the wide character written, or WEOF on error.

qsort

```
#include <stdlib.h>
void qsort( void *base, size_t n,
            size_t size, 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**.

This function is recursive, and therefore may need an increased user stack section!

raise

```
#include <signal.h>
int raise( int sig );
```

Sends the signal **sig** to the program.

See also "signal()".

Returns   zero if successful, or a non-zero value if unsuccessful.

rand

```
#include <stdlib.h>
int rand( void );
```

Returns   a sequence of pseudo-random integers, in the range 0 to RAND_MAX.

read

```
#include <unistd.h>
size_t  read( int fd, char * buffer, size_t count );
```

Reads a sequence of characters from a file. This function calls _read.

See also "_read()".

LIBRARIES

realloc

```
#include <stdlib.h>
void *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 7.3, *Heap Size*). By default no heap is allocated.

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.

remove

```
#include <stdio.h>
int remove( const char *filename );
```

Removes the named file, so that a subsequent attempt to open it fails.

Returns   zero if file is successfully removed, or
                a non-zero value, if the attempt fails.

rename

```
#include <stdio.h>
int rename( const char *oldname,
            const char *newname );
```

Changes the name of the file.

Returns   zero if file is successfully renamed, or
                a non-zero value, if the attempt fails.

rewind

```
#include <stdio.h>
void rewind( FILE *stream );
```

Sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. This function is equivalent to:

```
(void) fseek( stream, 0L, SEEK_SET );
clearerr( stream );
```

Returns   nothing.

scalloc

```
#include <stdlib.h>
void _shuge *scalloc( size_t nobj, size_t size );
```

Shuge variant of "calloc()". See section 7.3, *Heap Size*.

scanf

```
#include <stdio.h>
int scanf( const char *format, ... );
```

Performs a formatted read from the standard input stream.

See also "_read()".

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.
- The conversion characters `d`, `i`, `n`, `o`, `u` and `x` may be preceded by 'h' if the argument is a pointer to `short` rather than `int`, or by 'l' (letter ell) if the argument is a pointer to `long`. The conversion characters `e`, `f`, and `g` may be preceded by 'l' if a pointer `double` rather than `float` is in the argument list, and by 'L' if a pointer to 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 upper-case.
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.

Character	Scanned as

p	pointer; interpreted as a hexadecimal number, must be prefixed with: <near>, <far> or <huge>.
	
	For the different pointer types the following formats are expected:
	
	        <near> OOOO
	        <far> PPPP:OOOO
	        <huge> SS:OOOO
	where:
	
	        O is offset
	        P is page
	        S is segment

[...]	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 6-2: Scanf conversion characters

The 'p' conversion character can be used to read pointers. In the tiny and small memory models, pointers will be read as near pointers by default. In the medium and large memory models, pointers will be read as far pointers by default. By specifying one of the length modifiers 'h', 'l' or 'L', a pointer will always be read as near, far or huge respectively.

Two different version of the formatter (`_doscan.c`) are delivered. The LARGE version is able to scan everything as specified above. The SMALL version has no floating point scanning. When a floating point conversion character is found, `errno` is filled with the correct error number, scanf stops immediately.

Therefore the default formatter installed in the C library is the SMALL version.

setbuf

```
#include <stdio.h>
void setbuf( FILE *stream, char *buf );
```

Buffering is turned off for the `stream`, if `buf` is NULL.
Otherwise, setbuf is equivalent to:

   (void) setvbuf(stream, buf, _IOFBF, BUFSIZ)

Returns   nothing.

See also "setvbuf( )".

setjmp

```
#include <setjmp.h>
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` restores the current
              environment and jumps to this place with a non-zero return
              value.

See also "longjmp()".

setlocale

```
#include <locale.h>
char *setlocale( int category, const char *locale );
```

Selects the appropriate portion of the program's locale as specified by the
`category` and `locale` arguments.

Returns   the string associated with the specified `category` for the
              new locale if the selection can be honored.
              null pointer if the selection cannot be honored.

setvbuf

```
#include <stdio.h>
int setvbuf( FILE *stream, char *buf,
             int mode, size_t size );
```

Controls buffering for the `stream`; this function must be called before reading or writing. `mode` can have the following values:

_IOFBF   causes full buffering
_IOLBF   causes line buffering of text files
_IONBF   causes no buffering

If `buf` is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. `size` determines the buffer size.

Returns   zero if successful or a non-zero value for an error.

See also "setbuf( )".

sfree

```
#include <stdlib.h>
void sfree( void _shuge *p );
```

Deallocates the space pointed to by `p`. `p` Must point to space earlier allocated by a call to "scalloc()", "smalloc()" or "srealloc()". Otherwise the behavior is undefined.

Returns   nothing

LIBRARIES

signal

```
#include <signal.h>
void (*signal( int sig, void (*handler)(int)))(int);
```

Determines how subsequent signals will be handled. If `handler` is
SIG_DFL, the default behavior is used; if `handler` is SIG_IGN, the signal
is ignored; otherwise, the function pointed to by `handler` will be called,
with the argument of the type of signal. Valid signals are:

SIGABRT	abnormal termination, e.g. from `abort`
SIGFPE	arithmetic error, e.g. zero divide or overflow
SIGILL	illegal function image, e.g. illegal instruction
SIGINT	interactive attention, e.g. interrupt
SIGSEGV	illegal storage access, e.g. access outside memory limits
SIGTERM	termination request sent to this program

When a signal `sig` subsequenly occurs, the signal is restored to its default
behavior; then the signal-handler function is called, as if by
`(*handler)(sig)`. If the handler returns, the execution will resume
where it was when the signal occurred.

Returns   the previous value of `handler` for the specific signal, or
SIG_ERR if an error occurs.

sin

```
#include <math.h>
double sin( double x );
```

Returns   the sine of `x`.

sinh

```
#include <math.h>
double sinh( double x );
```

Returns   the hyperbolic sine of `x`.

smalloc

```
#include <stdlib.h>
void _shuge *smalloc( size_t size );
```

Shuge variant of "malloc()". See section 7.3, *Heap Size*.

sprintf

```
#include <stdio.h>
int sprintf( char *s, const char *format, ... );
```

Performs a formatted write to a string.

See also "printf()".

sqrt

```
#include <math.h>
double sqrt( double x );
```

Returns   the square root of **x**. $\sqrt{x}$, where $x \geq 0$.

srand

```
#include <stdlib.h>
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.

srealloc

```
#include <stdlib.h>
void _shuge *srealloc( void _shuge *p, size_t size );
```

Shuge variant of "realloc()". See section 7.3, *Heap Size*.

LIBRARIES

sscanf

```
#include <stdio.h>
int sscanf( char *s, const char *format, ... );
```

Performs a formatted read from a string.

See also "scanf()".

stat

```
#include <unistd.h>
int stat( const char * name, struct stat * buf );
```

Use the file system simulation feature of CrossView Pro to stat() a file on the host platform.

Returns   zero if successful,
              -1 on error.

strcat

```
#include <string.h>
char *strcat( char *s, const char *ct );
```

Concatenates string `ct` to string `s`, including the trailing NULL character.

Returns   s

strchr

```
#include <string.h>
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.

strerror

```
#include <string.h>
char *strerror( size_t n );
```

Returns   pointer to implementation-defined string corresponding to
               error n.

strftime

```
#include <time.h>
size_t strftime( char *s, size_t maxsize,
                 const char *format,
                 const struct tm *timeptr );
```

Formats date and time information from the structure *timeptr into s
according to the specified format format. format is analogous to a
printf format. Each %*c* is replaced as described below:

%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	local date and time representation
%d	day of the month (01-31)
%H	hour, 24-hour clock (00-23)
%I	hour, 12-hour clock (01-12)
%j	day of the year (001-366)
%m	month (01-12)
%M	minute (00-59)
%p	local equivalent of AM or PM
%S	second (00-59)

%U	week number of the year, Sunday as first day of the week (00-53)
%w	weekday (0-6, Sunday is 0)
%W	week number of the year, Monday as first day of the week (00-53)
%x	local date representation
%X	local time representation
%y	year without century (00-99)
%Y	year with century

%Z   time zone name, if any
%%   %

Ordinary characters (including the terminating '\0') are copied into **s**. No more than **maxsize** characters are placed into **s**.

Returns   the number of characters ('\0' not included), or
                 zero if more than **maxsize** characters where produced.

strlen

```
#include <string.h>
size_t strlen( const char *cs );
```

Returns   the length of the string in **cs**, not counting the NULL
                 character.

strncat

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

LIBRARIES

strncpy

```
#include <string.h>
char *strncpy( char *s, const char *ct, size_t n );
```

Copies string `ct` onto the string `s`, at most `n` characters are copied. Adds a trailing NULL character if the string is smaller than `n` characters.

Returns   s

strpbrk

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

strxfrm

```
#include <string.h>
size_t
strxfrm( char *ct, const char *cs, size_t n );
```

Transforms the string pointed to by `cs` and places the resulting string into
the array pointed to by `ct`. No more than `n` characters are placed into the
resulting string pointed to by `ct`, including the terminating null character.

Returns   the length of the transformed string.

swprintf

```
#include <wchar.h>
int swprintf( const wchar_t *s, size_t n,
              const wchar_t *format, ... );
```

Is equivalent to `fwprintf`, except that the output is written to an array of
wide characters (argument s). No more than `n` wide characters are written,
including a terminating null wide character.

Returns   the number of wide characters written in the array, not
               counting the terminating null wide character, or a negative
               value if an encoding error occurred or if `n` or more wide
               characters were requested to be written.

• • • • • • • • •

swscanf

```
#include <wchar.h>
int swscanf( const wchar_t *s,
             const wchar_t *format, ... );
```

Is equivalent to `fwscanf`, except that the input is obtained from a wide string (argument s).

Returns   the number of input items assigned or EOF on error.

tan

```
#include <math.h>
double tan( double x);
```

Returns   the tangent of `x`.

tanh

```
#include <math.h>
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 in seconds, or -1 if the time is not available.

tmpfile

```
#include <stdio.h>
FILE *tmpfile( void );
```

Creates a temporary file of the mode "wb+" that will be automatically
removed when closed or when the program terminates normally.

Returns   a stream if successful, or NULL if the file could not be
                created.

tmpnam

```
#include <stdio.h>
char *tmpnam( char s[L_tmpnam] );
```

Creates a temporary name (not a file). Each time tmpnam is called a
different name is created.
tmpnam(NULL) creates a string that is not the name of an existing file,
and returns a pointer to an internal static array. tmpnam(s) creates a
string and stores it in s and also returns it as the function value. s must
have room for at least L_tmpnam characters. At most TMP_MAX different
names are guaranteed during execution of the program.

Returns   a pointer to the temporary name, as described above.

toascii

```
#include <ctype.h>
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>
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>
int toupper( int c );
```

Returns   c converted to an uppercase character if it is a lowercase character, otherwise c is returned.

towctrans

```
#include <wctype.h>
wint_t towctrans( wint_t wc, wctrans_t desc );
```

Returns   the mapped value of wc using the mapping described by desc.

For example, the function tolower(wc) is the same as specifying:

```
towctrans(wc, wctrans("tolower"))
```

towlower

```
#include <wctype.h>
wint_t towlower( wint_t wc );
```

Returns   wc converted to a lowercase wide character if it is an uppercase wide character, otherwise wc is returned.

towupper

```
#include <wctype.h>
wint_t towupper( wint_t wc );
```

Returns wc converted to an uppercase wide character if it is a
             lowercase wide character, otherwise wc is returned.

ungetc

```
#include <stdio.h>
int ungetc( int c, FILE *fin );
```

Pushes at the most one character back onto the input buffer.

Returns EOF on error.

ungetwc

```
#include <wchar.h>
wint_t ungetwc( wint_t c, FILE *stream );
```

Pushes at the most one wide character back onto the input stream.

Returns the wide character pushed back, or WEOF on error.

unlink

```
#include <unistd.h>
int unlink( const char * name );
```

Removes the named file, so that a subsequent attempt to open it fails. This
function calls _unlink.

Returns zero if file is successfully removed, or
             a non-zero value, if the attempt fails.

va_arg

```
#include <stdarg.h>
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>
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>
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>
int vfprintf( FILE *stream,
              const char *format, va_list arg );
```

Is equivalent to vprintf, but writes to the given stream.

See also "vprintf()" and "_write()".

vprintf

```
#include <stdio.h>
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()" and "_write()".

vsprintf

```
#include <stdio.h>
int vsprintf( char *s, const char *format,
              va_list arg );
```

Does a formatted write to a string. Instead of a variable argument list as for
printf(), this function expects a pointer to the list.

See also "printf()" and "_write()".

vfwprintf

```
#include <wchar.h>
int vfwprintf( FILE *stream,
               const wchar_t *format, va_list arg );
```

Is equivalent to `fwprintf`, except that instead of a variable argument list
this function expects a pointer to the list.

Returns   the number of wide characters transmitted, or a negative
                 value if an output or encoding error occurred.

vswprintf

```
#include <wchar.h>
int vswprintf( const wchar_t *s, size_t n,
               const wchar_t *format, va_list arg );
```

Is equivalent to `swprintf`, except that instead of a variable argument list this function expects a pointer to the list.

Returns   the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if an encoding error occurred or if `n` or more wide characters were requested to be written.

vwprintf

```
#include <wchar.h>
int vwprintf( const wchar_t *format, va_list arg );
```

Is equivalent to `wprintf`, except that instead of a variable argument list this function expects a pointer to the list.

Returns   the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

wcrtomb

```
#include <wchar.h>
size_t wcrtomb( char *s, wchar_t wc, mbstate_t *ps );
```

Determines the number of bytes needed to represent the multi-byte character that corresponds to the wide character given by `wc` (including any shift sequences). It stores the multi-byte character representation in the array pointed to by `s` (if `s` is not a null pointer). At most MB_CUR_MAX characters are stored. If `wc` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns   the number of bytes, or `(size_t)-1` if the value of `wc` does not correspond to a valid wide character.

wcscat

```
#include <wchar.h>
wchar_t *wcscat( wchar_t *s1, const wchar_t *s2 );
```

Concatenates a copy of wide string `s2` to string `s1`, including the trailing null wide character. The initial wide character of `s2` overwrites the null wide character at the end of `s1`.

Returns   `s1`

wcschr

```
#include <wchar.h>
wchar_t *wcschr( const wchar_t *s, wchar_t c );
```

Returns   a pointer to the first occurrence of wide character `c` in the wide string `s`. If not found, NULL is returned.

wcscmp

```
#include <wchar.h>
int wcscmp( const wchar_t *s1, const wchar_t *s2 );
```

Compares wide string `s1` to wide string `s2`.

Returns   `<0` if `s1` < `s2`,
                `0`   if `s1` == `s2`,
                `>0` if `s1` > `s2`.

wcscoll

```
#include <wchar.h>
int wcscoll( const wchar_t *s1, const wchar_t *s2 );
```

Compares wide string `s1` to wide string `s2`. The comparison is based on wide strings interpreted as appropriate to the program's locale.

Returns   `<0` if `s1` < `s2`,
                `0`   if `s1` == `s2`,
                `>0` if `s1` > `s2`.

wcscpy

```
#include <wchar.h>
wchar_t *wcscpy( wchar_t *s1, const wchar_t *s2 );
```

Copies wide string `s2` intto wide string `s1`. including the trailing null wide character.

Returns   s1

wcscspn

```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2 );
```

Returns   the length of the maximum initial segment of wide string `s1` which consists entirely of wide characters *not* from wide string `s2`.

wcsftime

```
#include <wchar.h>
size_t wcsftime( wchar_t *s, size_t maxsize,
                 const wchar_t *format,
                 const struct tm *timeptr );
```

This function is equivalent to the `strftime` function, except that:

-   The argument `s` points to the initial element of an array of wide characters into which the generated output is to be placed.
-   The argument `maxsize` indicates the limiting number of wide characters.
-   The argument `format` is a wide string and the conversion specifiers are replaced by corresponding sequences of wide characters.

Returns   the number of wide characters ('\0' not included), or zero if more than `maxsize` wide characters where produced.

LIBRARIES

wcslen

```
#include <wchar.h>
size_t wcslen( const wchar_t *s );
```

Returns   the length of the wide string in **s**, not counting the null wide
                character.

wcsncat

```
#include <wchar.h>
wchar_t *wcsncat( wchar_t *s1, const wchar_t *s2,
                  size_t n );
```

Concatenates at most **n** wide characters from wide string **s2** to wide string
s1. A terminating null wide character is always appended to the result.

Returns   s1

wcsncmp

```
#include <wchar.h>
int wcsncmp( const wchar_t *s1, const wchar_t *s2,
             size_t n);
```

Compares at most **n** wide characters of wide string **s1** to wide string **s2**.

Returns   <0   if **s1** < **s2**,
               0   if **s1** == **s2**,
               >0   if **s1** > **s2**.

wcsncpy

```
#include <wchar.h>
wchar_t *wcsncpy( wchar_t *s1, const wchar_t *s2,
                  size_t n );
```

Copies at most **n** characters of wide string **s2** onto the wide string **s1**.
Adds trailing null characters if the string is smaller than **n** wide characters.

Returns   s1

wcspbrk

```
#include <wchar.h>
wchar_t *wcspbrk( const wchar_t *s1,
                  const wchar_t *s2 );
```

Returns   a pointer to the first occurrence in `s1` of any wide character out of wide string `s2`. If none are found, NULL is returned.

wcsrchr

```
#include <wchar.h>
wchar_t *wcsrchr( const wchar_t *s, wchar_t c );
```

Returns   a pointer to the last occurrence of `c` in the wide string `s`. If not found, NULL is returned.

wcsrtombs

```
#include <wchar.h>
size_t wcsrtombs( char *dst, const wchar_t **src,
                  size_t len, mbstate_t *ps );
```

Converts a sequence of wide characters from the array indirectly pointed to by `src` into a sequence of corresponding multi-byte characters that begins in the conversion state described by the object pointed to by `ps`. This function then stores these multi-byte characters into the array pointed to by `dst`, stopping if a multi-byte character would exceed the limit of `len` total bytes, or when a wide character is reached that does not correspond to a valid multi-byte character, or if a null character is stored.

Returns   the number of bytes modified (not including a terminating null character, if any), or (`size_t`)-1 if a wide character is encountered that does not correspond to a valid multi-byte character.

LIBRARIES

wcsspn

```
#include <wchar.h>
size_t wcsspn( const wchar_t *s1, const wchar_t *s2 );
```

Returns   the length of the maximum initial segment of wide string `s1` which consists entirely of wide characters from wide string `s2`.

wcsstr

```
#include <wchar.h>
wchar_t *wcsstr( const wchar_t *s1,
                 const wchar_t *s2);
```

Returns   a pointer to the first occurrence of wide string s2 in the wide string `s1`. Returns NULL if not found.

wcstod

```
#include <wchar.h>
double wcstod( const wchar_t *nptr,
               wchar_t **endptr );
```

Converts the initial portion of the wide string pointed to by `nptr` to `double`. Initial white spaces are skipped. A pointer to the final wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Returns   the converted value, or zero if no conversion could be performed.

wcstok

```
#include <wchar.h>
wchar_t *wcstok( wchar_t *s1, const wchar_t *s2,
                 wchar_t **ptr );
```

Searches the wide string `s1` for tokens delimited by wide characters from wide string `s2`. It terminates the token with a null character.

Returns   a pointer to the first wide character of a token.
A subsequent call with `s1` == NULL will return the next token in the string.

wcstol

```
#include <wchar.h>
long int wcstol( const wchar_t *nptr,
                 wchar_t **endptr, int base );
```

Converts the initial portion of the wide string pointed to by `nptr` to `long int`. 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. A pointer to the final wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Returns   the converted value, or zero if no conversion could be performed.

wcstombs

```
#include <stdlib.h>
size_t wcstombs( char *s, const wchar_t *pwcs,
                 size_t n );
```

Converts a sequence of wide characters from the array pointed to by
`pwcs`, into a sequence of multi-byte characters that begins in the initial
shift state and stores these multi-byte characters into the array pointed to
by `s`, stopping if a multi-byte character would exceed the limit of `n` total
bytes or if a null character is stored.

Returns   the number of bytes modified (not including a terminating
                null character, if any), or (`size_t`)-1 if a wide character is
                encountered that does not correspond to a valid multi-byte
                character.

wcstoul

```
#include <wchar.h>
unsigned long int wcstoul( const wchar_t *nptr,
                  wchar_t **endptr, int base );
```

Same as `wcstol`, except that it converts the initial portion of the wide
string to `unsigned long int`.

Returns   the converted value, or zero if no conversion could be
                performed.

wcsxfrm

```
#include <wchar.h>
size_t wcsxfrm( wchar_t *s1, const wchar_t *s2,
                size_t n);
```

Transforms the wide string pointed to by `s2` and places the resulting string
into the array pointed to by `s1`. No more than `n` wide characters are
placed into the resulting array pointed to by `s1`, including the terminating
null wide character.

Returns   the length of the transformed wide string.

wctob

```
#include <wchar.h>
int wctob( wint_t c );
```

Determines whether **c** corresponds to a member of the extended character set whose multi-byte character representation is a single byte when in the initial shift state.

Returns   EOF if **c** does not correspond to a multi-byte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character as an **unsigned char** converted to an **int**.

wctomb

```
#include <stdlib.h>
int wctomb( char *s, wchar_t wchar );
```

Determines the number of bytes needed to represent the multi-byte character corresponding to the wide character whose value is **wchar** (including any change in the shift state). It stores the multi-byte character representation in the array pointed to by **s** (if **s** is not a null pointer). At most MB_CUR_MAX characters are stored. If the value of **wchar** is zero, the wctomb function is left in the initial shift state.

Returns   the number of bytes, or -1 if the value of **wchar** does not correspond to a valid multi-byte character.

wctrans

```
#include <wctype.h>
wctrans_t wctrans( const char *property );
```

Constructs a value with type **wctrans_t** that describes a mapping between wide characters identified by the string argument **property**. Valid strings are: **tolower** or **toupper**.

See also "towctrans()".

Returns   a non-zero value that is valid as the second argument to the **towctrans** function, if **property** identifies a valid mapping of wide characters; otherwise, it returns zero.

LIBRARIES

wctype

```
#include <wctype.h>
wctype_t wctype( const char *property );
```

Constructs a value with type `wctype_t` that describes a class of wide characters identified by the string argument `property`. Valid strings are: `alnum`, `alpha`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper` or `xdigit`.

See also "iswctype()".

Returns   a non-zero value that is valid as the second argument to the `iswctype` function, if `property` identifies a valid class of wide characters; otherwise, it returns zero.

wmemchr

```
#include <wchar.h>
wchar_t *wmemchr( const wchar_t *s,
                  wchar_t c, size_t n );
```

Checks the first `n` wide characters of `s` on the occurrence of wide character `c`.

Returns   a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

wmemcmp

```
#include <wchar.h>
int wmemcmp( const wchar_t *s1,
             const wchar_t *s2, size_t n );
```

Compares the first `n` wide characters of `s1` to the first `n` wide characters of `s2`.

Returns   <0 if `s1` < `s2`,
             0   if `s1` == `s2`,
             >0 if `s1` > `s2`.

wmemcpy

```
#include <wchar.h>
wchar_t *wmemcpy( wchar_t *s1,
                  const wchar_t *s2, size_t n );
```

Copies **n** wide characters from **s2** to **s1**. Does not check for memory overlapping.

Returns   s1

wmemmove

```
#include <wchar.h>
wchar_t *wmemmove( wchar_t *s1,
                   const wchar_t *s2, size_t n );
```

Copies **n** wide characters from **s2** to **s1**. Overlapping objects will be handled correctly.

Returns   s1

wmemset

```
#include <wchar.h>
wchar_t *wmemset( wchar_t *s, wchar_t c, size_t n );
```

Fills the first **n** wide characters of **s** with the value of **c**.

Returns   s

wprintf

```
#include <wchar.h>
int wprintf( const wchar_t *format, ... );
```

Is equivalent to **fwprintf**, except that the output is written to **stdout** instead of a stream.

Returns   the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

LIBRARIES

wscanf

```
#include <wchar.h>
int wscanf( const wchar_t *format, ... );
```

Is equivalent to `fwscanf`, except that the input is obtained from `stdin`.

Returns   the number of input items assigned or EOF on error.

write

```
#include <unistd.h>
size_t  write( int fd, char * buffer, size_t count );
```

Write a sequence of characters to a file. This function calls _write.

See also "_write()".

6.7   CAN LIBRARY INTERFACE DESCRIPTION

check_busoff_16x

```
#include <can_ext.h>
unsigned char check_busoff_16x( void );
```

Check if a bus off situation has occurred and recover from bus off.

Returns   one if CAN controller was in bus off state, zero otherwise.

check_mo_16x

```
#include <can_ext.h>
unsigned char check_mo_16x( unsigned char nr );
```

Check for new data in a message object.

Returns   one if the specified message object contains new date, zero
                   otherwise.

check_mo15_16x

```
#include <can_ext.h>
unsigned char check_mo15_16x( void );
```

Check for new data or remote frame in message object 15.

Returns   one if message object 15 contains new data, zero otherwise.

def_mo_16x

```
#include <can_ext.h>
void def_mo_16x( unsigned char nr, unsigned char xtd,
                 unsigned long id, unsigned char dir,
                 unsigned char dlc, unsigned char txie,
                 unsigned char rxie );
```

Define a message object in the CAN module.

Returns   nothing.

init_can_16x

```
#include <can_ext.h>
void init_can_16x( unsigned int baud_rate,
                   unsigned char eie,
                   unsigned char sie,
                   unsigned char ie );
```

Initialization of the CAN module.

Returns   nothing.

ld_modata_16x

```
#include <can_ext.h>
void ld_modata_16x( unsigned char nr,
                    unsigned char * upl_data_ptr );
```

Load the data bytes of a message object.

Returns   nothing.

rd_modata_16x

```
#include <can_ext.h>
void rd_modata_16x( unsigned char nr,
                    unsigned char * downl_data_ptr );
```

Read the data bytes of a message object.

Returns   nothing.

rd_mo15_16x

```
#include <can_ext.h>
void rd_mo15_16x( unsigned char * mo15_db_ptr,
                  unsigned long * mo15_id_ptr,
                  unsigned char * mo15_dlc_ptr );
```

Read the contents of message object 15.

Returns   nothing.

• • • • • • • • •

send_mo_16x

```
#include <can_ext.h>
void send_mo_16x( unsigned char nr );
```

Send message object.

Returns   nothing.

6.8 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. These files are placed in the directory `lib\src` (Windows) or `lib/src` (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 '**ar166 t c166m.lib**', for example.

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 '**ar166 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.

You can rebuild your library with **mk166**. To use the correct makefile, first make sure you are in the directory of the library you want to rebuild: `lib\src\`*architecture*`\`*library* (Windows) or `lib\src\`*architecture*`\`*library* (UNIX). Use **mk166** to rebuild your library now. (You may want to make a backup copy of the original library first.)

• • • • • • • • •

CHAPTER

7

**RUN–TIME
ENVIRONMENT**

TASKING

CHAPTER

7
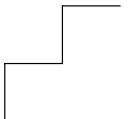
7.1 STARTUP CODE

When linking (Task Concept) or locating (Flat Interrupt Concept) the module containing `main()` which is an object module containing the C startup code has to be linked to the application. This module, called `start.obj`, is included in each C library with a system startup configuration default for the library it is included in. The compiler generates a reference to this module when it translates the definition of the `main()` function. This reference causes the `start.obj` to be extracted from the library by **l166**.

This file specifies the run-time environment of your C166 application. The file is delivered in assembly source (`start.asm`) in the directory `lib\src`. The file `start.asm` includes the file `cstartx.asm` or `cstartx2.asm` depending on the selected architecture. Modifications to these files are not necessary since all parameters can be manipulated using macro preprocessor symbols.

Startup code and EDE

When you use EDE, the startup code will be automatically generated and included in the project. The contents of the EDE generated startup code is largely defined by the options you set in EDE. When you want to use your own startup code you can disable the generation of the startup code in EDE:

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

   The Project Options dialog appears.

2. Expand the **Application** entry and select **Startup**.

3. Disable the check box **Generate system startup code and add it to project**.

4. Click **OK**.

5. Remove the file `start.asm` from the project and add your own startup code.

If necessary you can specify EDE to generate the startup code in a different file than `start.asm`:

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

   The Project Options dialog appears.

2. Expand the **Application** entry and select **Startup**.

3. Enable the check box **Generate system startup code and add it to project**.

4. Specify the filename in the **Startup code file name** field.

5. Click **OK**.

    After this multiple startup code files may be present in your project.

6. Manually remove the obsolete startup code files from the project.

You can control the contents of the generated startup code from the **Startup** entry in the Project Options dialog. From the subentries under **Startup** you can specify the registers and their settings that must be known to the startup code: enable the **Include in startup code** check box for the register settings you want to add to the startup code. EDE automatically generates the register initializations in the startup code.

Startup code and the command line

When you are not using EDE, you must use **m166** before **a166** when a new version of the object file has to be created:

```
m166 start.asm DEFINE( MODEL, LARGE )
a166 start noprint
```

You must specify the memory model for the preprocessing phase. Therefore you have to define the preprocessor symbol MODEL. You can do this with the **m166** command line control DEFINE by defining the memory model you are using. When preprocessing the startup file, MODEL is checked to select, skip or include certain pieces of code.

The new `start.obj` can be supplied to **l166** when linking the module containing `main()`. **l166** will use this object instead of the object from the library.

Preprocessor symbols used in startup code

There are a number of other preprocessor symbols used, which can be enabled or disabled using the command line control DEFINE (Syntax: DEFINE(*identifier* [, *replacement*])).

RUN–TIME

In the startup file the following preprocessor symbols are used (please also review `cstartx.asm` or `cstartx2.asm`):

EX_AB   Must be enabled (set to 1) if the C library function `exit()` or `abort()` is called by the application. Otherwise it must be cleared (set to 0). Default cleared, because the total code size is increased, due to assumptions about buffered file I/O, which must be flushed at exit.

_EXT2   Must be enabled (set to 1) when a XC16x/Super10 architecture (ext2) needs to be initialized. It must be cleared (set to 0) when a C16x/ST10 architecture needs to be initialized.

BIT_INIT   Must be enabled (set to 1) if initialized bit variables (`bit b = 1;`) are used, so the initialization is done at startup. Non-initialized bit variables are always set to 0. Default set to 0, because initialized bit variables are very seldom used and rather expensive in both ROM space and execution time during startup. Therefore, if possible, initialized bit variables should be avoided.

NOBITCLEAR
      When set, skips clearing of the bitaddressable RAM.

EVA   Must be enabled (set to 1) when using the ROM/RAM monitor on evaluation boards as execution environment. Needed to force the tiny model to execute with the CPU segmentation enabled and to prevent the startup code to clear the bit-addressable area, which contains monitor data. It also starts the application with interrupts enabled and provides CrossView Pro with information about the configuration when the C167 is used. Default enabled.

_CPU   Must be set when one of the following processoris used: 165-UTAH (165Utah), 167CS-40 (167CS40), SDA6000 (sda6000). Set the _CPU symbol to the appropriate value mentioned between brackets. If none of these derivatives is used then _CPU does not need to be set.

_USRSTACK
      Must be enabled (set to 1) to support the user stack model. Default disabled. See section 3.2.2, *User Stack Model* for more details.

• • • • • • • • •

CALLINIT   Can be set to a function to be called before `main`. This
                function may not have any return value and may not have
                any arguments. This function can be used, for example, to
                initialize the serial port before `main` is called. This is useful
                for building benchmark programs without making any
                modifications to the original source.

CALLEINIT

                Can be set to a function to be called before the EINIT
                instruction is executed, but after register initialization. Like
                the CALLINIT function, it may not have a return value or any
                arguments.

CALL_USER

                Can be set to an include file containing the `_main` label
                entry.

SSKENABLE

                If set, intializes the system stack for XC16x/Super10
                architectures using a modifiable SYSSTACK system.

__SSKSIZE   Determines the stack size in bytes on XC16x/Super10
                architectures.

__SSKSEG   Determines the segment where the system stack is positioned
                on XC16x/Super10 architectures.

In the startup file a code section named __CSTART_PR is declared. In this
code section the task procedure __CSTART is declared, using interrupt
number 0, which is the power-on vector of the processor.

First the system is configured using a macro for each configuration item:
wait states, read/write signal delay, system clock output, segmentation
control, system stack size etc. You must specify these values using the
appropiate macros, depending on the specific needs of your target system.
Please review the appropriate startup file for an exact overview of
initialized registers and macros used.

The system stack registers (overflow, underflow and stack pointer) are
initialized using the predefined symbols ?SYSSTACK_BOTTOM and
?SYSSTACK_TOP. The assembler, linker and locator treat predefined
symbols (all starting with a '?') in a special way. They give the assembler
programmer access to information which is normally not available before
the locate stage of the application.

RUN–TIME

After the context pointer register is set to the register bank of this task, write output is enabled and the 'end-of-initialization' instruction (EINIT) is executed.

All bit addressable memory is cleared, because this guarantees all non initialized bit variables of each task to have the value of 0.

The startup code also takes care of initialized static/public C variables of each task, residing in the different RAM areas (not `const` or `#pragma romdata`). All these initialized variables are allocated in both a ROM and RAM section for each category (bit, near, far, huge). See section 3.8, *Initialized Variables*, for more details. The startup code copies the initial values of initialized C variables <u>for the whole application</u> (all tasks in the Task Concept) from ROM to RAM using a table (in the global C166_INIT section), which has been built by the compiler. The predefined symbol ?C166_INIT_HEAD contains the start address of the table.

In ANSI-C all non-initialized static/public C variables must have the initial value of 0. Non-initialized bit variables are already cleared by previous code. Therefore, the startup code clears the non-initialized non-bit variables <u>of the whole application</u> (all tasks in the Task Concept) using a table (in the global C166_BSS section), which has been built by the compiler (unless the **–Ob** option has been used). The predefined symbol ?C166_BSS_HEAD contains the start address of the table. See section 3.9, *Non-Initialized Variables*, for more information.

Finally, the DPP registers are initialized, depending on the memory model used. DPP0 to DPP2 are initialized accordingly. The predefined symbols ?BASE_DPP0 to ?BASE_DPP2 are used to initialize DPP0 to DPP2.

Last but not least, the user stack pointer is initialized using the predefined symbol ?USRSTACK_TOP.

When everything described above has been executed, your C application is called, using the public label _main, which has been generated by **c166** for the C function `main()`. When the C application 'returns', which is not likely to happen in an embedded environment, you can specify if the program uses the function `exit( )`, `abort( )` or `atexit( )`.

At the assembly label __EXIT, the system stack pointer, the user stack pointer and the floating point stack (if floats are used) are restored and the program performs an endless loop setting the CPU in power down mode (IDLE instruction).

• • • • • • • • •

7.2 STACK SIZE

c166 maintains two types of stack: the system stack and the user stack.

The *system stack* is used for return addresses (CALL/RET instructions) and can be accessed via PUSH/POP instructions (using the SP register). Because the system stack is very small (internal memory for the C166/ST10), **c166** tries to avoid it as much as possible. Code generator temporaries are pushed on the user stack. Via the **–Ou** option it is even possible to let a task switch (interrupt) use the user stack instead of the system stack. As described above, you must specify the size of the system stack size in the system startup code (SYSCON register), which is the system stack size for all tasks (the whole application).

For XC16x/Super10 architectures, the system stack size is determined by specifying a SYSSTACK section of the required size and using the SSKDEF 7 directive option. The locator can relocate this section. Use the preprocessor macros SSKENABLE, __SSKSEG and __SSKSIZE to determine the correct system stack.

From EDE you can set the system stack size in the **Stack and Heap** page of the **Linker/Locator** entry in the **Project | Project Options** dialog.

If **–P** is used, the system stack is not used at all. See section 3.2.2, *User Stack Model* for details.

The *user stack* is the so-called 'C stack'. **c166** uses R0 as 'User Stack Pointer' and the [-R0]/[R0+] addressing modes perform push/pop sequences. If data paging is used (medium and large memory model), the user stack is limited to 16K (one page). In these models, **c166** uses DPP1 as 'user-stack page number'. The locator combines the user stack areas of each task to one global user stack area (with cumulated size). A context switch inherits the user stack pointer (R0) value in the new register bank and DPP1 remains unchanged.

c166 estimates the needed user stack size for each C module by adding the stack sizes of each function to each other. This amount of bytes is allocated in the data section called C166_US (see section 3.2.3, *Section Allocation*). However, in most cases this is too big, because not all functions are active simultaneously. In other cases, the size will be too small, e.g. when recursive functions are present (note that `qsort()` is implemented as a recursive function).

You can modify the user stack size using the SECSIZE control of the locator.

RUN–TIME
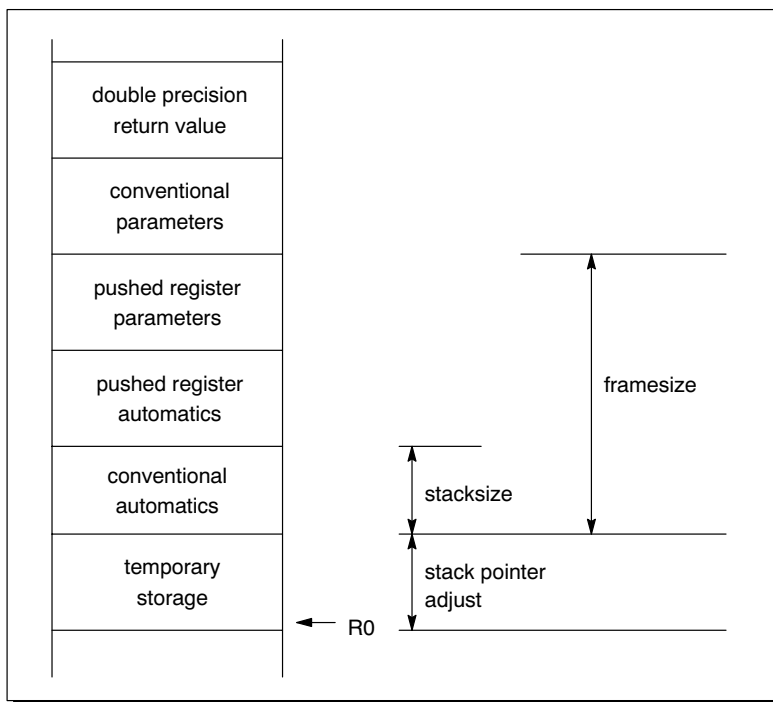
Figure 7-1: Stack diagram

Example:

```
l166 task t1.lno SECSIZE(C166_US(-50))
     task t2.lno SECSIZE(C166_US(-10))
     TO   applic.out
```

7.3   HEAP SIZE

The heap is only needed when dynamic memory management library functions are used: `malloc()`, `calloc()`, `free()` and `realloc()`. The heap is allocated by the linker <u>for each task</u> in a special (public) section called ?C166_FHEAP or ?C166_NHEAP, both with the class name ?CHEAP having a default size of 0 bytes. If you are using one of the memory allocation functions listed above in a certain task, you must change the heap size <u>for that task</u> using the `HEAPSIZE` control at link stage.

When the Flat Interrupt Concept is used the link stage is skipped and the locator generates the ?C166_NHEAP and ?C166_FHEAP sections when it is needed. You can use the `HEAPSIZE` control for changing the heap size at locate stage. The dynamic memory management library functions are not reentrant, because they use static data for the memory management. This means that when the memory management functions are used, it is not possible to interrupt them with an interrupt function which also uses the memory management functions. If reentrancy is needed with memory management functions, you should use the Task Concept where each interrupt can have its own memory management.

In the tiny and small model the default memory allocation routines use the ?C166_NHEAP section, which has the section type 'LDAT' allowing a total heap size up to 64K. Because paging is not used (except for the small SND variant, a linear 16-bit pointer is returned), the maximum amount of memory asked for is not limited to a page (16K).

In the medium and large model the ?C166_FHEAP section is used by the default memory allocation routines. This section has the section type 'HDAT' allowing a total heap size greater then 64K. However, in these models paging is used: a far pointer is returned. This means that you cannot allocate (dynamically) a single buffer greater than one page (16K). Of course you can allocate the whole heap in pieces of (approximately) 16K. In these models, you should use memory allocation with great care, because the paging approach may introduce 'fragmentation' of the heap. For example, if you allocate two times 9K of memory, the second request does not fit in the same page as the first 9K. So, 9K will be allocated in the next page, introducing a gap of approximately 7K, which only will be used for requests fitting in 7K.

In the huge memory model, the ?C166_FHEAP section is used by default. In this model, a huge pointer is returned, allowing allocation of objects larger than 64K (in fact, as large as the total heap).

It is possible to use various flavours of the memory allocation routines. The following table shows the available variants and indicates in which model these variants are used by default:

Routine variant	Pointer type	Used by default in	Size
nmalloc ncalloc nrealloc nfree	_near	tiny, small	< 64kB
fmalloc fcalloc frealloc ffree	_far	medium, large	< 16kB
smalloc scalloc srealloc sfree	_shuge		< 64kB
hmalloc hcalloc hrealloc hfree	_huge	huge	< 16MB

Please note that the non-near variants of the memory allocation routines all use the same heap stack ?C166_FHEAP and are in the same source module. Use of the **fmalloc()** routine will therefore automatically include support for the **smalloc()** and **hmalloc()** routines.

7.4 ASSEMBLY LANGUAGE INTERFACING

Assembly language functions can be called from C and vice versa. The names used by **c166** are case sensitive, so you must specify **a166** to act case sensitive too, using the $CASE control. **c166** adds an underscore for the name of public C variables, to distinguish these names from the C166/ST10 registers. So, any names used or defined in C for the C166/ST10 must have a leading underscore in assembly code.

In section 3.15, *Register Usage* of the chapter *Language Implementation*, the registers used for return values of functions are explained. Note that R0 is used as user stack pointer and must be used in the assembly function accordingly. If fast parameter passing is used with this assembly function or functions called by this assembly function, R12 to R15 can not be used as scratch registers. Note that if you want to use one of the registers R6 to R9, you must save it on the user stack at entry and restore at exit, because this register might contain a C register variable of another C function. Registers R1-R5, R10 and R11 are free.

In section 3.6, *Function Parameters* of the chapter *Language Implementation* is described how parameter passing is supported by **c166**. If you do not want parameter passing in registers (e.g. existing assembly function expecting parameters on the user stack) you must use the keyword `stackparm` (as function qualifier) in the full C prototype of the assembly language function. The quickest (and most reliable) way to make an assembly language function, which must conform to C for the C166/ST10, is to make the body of this function in C, and compile this module. 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):

```
int stackparm
assem( char c, int i )
{
    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.

Inline assembly

A second method to create an interface to assembly is to use inline
assembly in C. Assembly lines in the C source must be introduced by a
'#pragma asm', the end is indicated by a '#pragma endasm'.

For example:

```
int
inline( char c, int i )
{
        int j = i - c;

        if ( j > 5 )
        {
#pragma asm
                NOP  ; do something in assembly
#pragma endasm
                j = 0;
        }
        return ( j );
}
```

If the inserted assembly code does not change any registers, like in the
example above, also '#pragma asm_noflush' may be used instead of
'#pragma asm'. The advantage of this pragma is that the peephole buffer is
not flushed, so the compiler will emit a JMPR instructions instead of a
JMPA instruction for the condition above. Note that the inserted assembly
is NOT interpreted, so code size reported is only the code generated for C
statements. The disadvantage of the '#pragma asm_noflush' is that the
distance checking for relative jumps becomes your responsibility !

Note that the compiler also does NOT recognize inline CALL instructions.
If a function does not call any other function from C, it is treated like a
'leaf' function, so parameter registers of this function are not saved on the
user stack at function entry. If a 'leaf' function calls another function using
inline assembly, it is your responsibility to preserve the parameter registers
(if any) of this 'leaf' function.

Global constants in different modules

Proper interfacing often requires global constants in different modules, which can be a problem when assembly and C modules are mixed. You can work around this problem by using the C preprocessor on assembly files before assembling. For example:

```
header.h:
        /* comment */
        #define CONSTANT 0x1

module1.c:
        #include "header.h"
        ...
        int c=CONSTANT;

module2.asm:
        #include "header.h"
        ...
        MOV R0, CONSTANT
```

The assembler understands the C notation for hexadecimal numbers, so no special conversions are needed for that. The modules are built using:

```
c166 module1.c -o module1.src
a166 module1.src TO module1.obj
```

and

```
c166 -E -o module2.src module2.asm
a166 module2.src TO module2.obj
```

When you use the control program, the default preprocessor for assembly files is **m166**. You can change this default by using the **-cprep** option, which forces the control program to use the C preprocessor instead. The above files can thus be easily built with the following single command:

```
cc166 module1.c module2.asm -cprep -c
```

Please note that the C preprocessor will not replace anything between #pragma asm and #pragma endasm in your C source.

RUN-TIME

APPENDIX

A

MISRA C

TASKING

APPENDIX

A
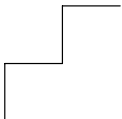
Supported and unsupported MISRA C rules

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

MISRA C

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

MISRA C

	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 3.19, *C Code Checking: MISRA C*, in Chapter *Language Implementation*.

MISRA C

**DEBUG
ENVIRONMENT**

TASKING
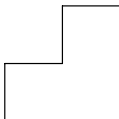
1 CROSSVIEW PRO AND EVALUATION BOARDS

When you use an evaluation board with CrossView Pro, a monitor will be run from the memory where your application is loaded and running. You should use the **1166** RESERVE MEMORY locator control to prevent the locator from locating sections in the memory areas in use by the monitor. For example:

    RESERVE MEMORY(0FD00h to 0FD4Bh)

Please see the CrossView user's manual which areas are in use by the monitor that is used for your evaluation board.

In the `start.asm` file, the @EVA symbol must be enabled (set to 1).

When using a ROM monitor with a dual vector table, the vector table of your application should be located at the memory location where the monitor expects it to be. Use the **1166** VECTAB locator control to supply the vector table start location to the locator. For example:

    VECTAB(08000h)

Please refer to the CrossView user's manual for the required vector table location for the board and monitor that you use.

When using this dual vector table ROM monitor, you must also supply the **-s***startaddress* option to the **ieee166** IEEE-695 object formatter. The *startaddress* should be address where you located the vector table with the VECTAB control. This address will be generated in the absolute file. CrossView Pro will start execution at this address after a program reset.

2 KONTRON DEBUGGER

When using Kontron debuggers, the following operation remarks exist:

- Use the TASKING **ieee166** converter program to generate an IEEE-695 output file from the absolute (located) output file. The Kontron **KSE695** filter program is needed to translate this IEEE-695 file into Kontron object and symbol files.

- You can use the compiler option **–g** to generate debug information for use by Kontron debuggers. Versions of **KSE695** previous to v4.3 (04) may require using the compiler option **–gb**. The **–gb** option prevents **c166** from emitting 'bit', 'bitfield' and '80166 pointer behavior' high level language information.

- The **KSE695** command line option '**–t t –x .**' must be used when converting IEEE-695 format to Kontron format.

  –t t = Specify TASKING **c166** IEEE-695 format.

  –x   = Preserve filename and extension information found in the IEEE-695 file.

- Kontron debuggers supports all high level language debug information generated by **c166**.

- Kontron debuggers support debugging of TASKING **a166** assembly files at the source code level. You can use the Kontron **LINE166** utility before preprocessing source with TASKING **m166** or assembling with **a166**.

The **LINE166** utility has the following command line syntax:

```
LINE166  inputfile  outputfile
```

where, *inputfile* is the file you would normally process with the TASKING macro preprocessor or assembler and *outputfile* is the instrumented output file. This output file is the file you must use for preprocessing/assembly. The input and output filename must differ.

DEBUG ENVIRONMENT

Batch file when TASKING **m166** used   Batch file when **m166** not used

```
@echo off                              @echo off
rem RELINE1.BAT                        rem RELINE2.BAT
line166 %1.asm %1.a66                  line166 %1.asm %1.a66
if errorlevel 1 goto end               if errorlevel 1 goto end
m166 %1.a66                            a166 %1.a66 debug
if errorlevel 1 goto end               if errorlevel 1 goto end
del %1.a66                             del %1.a66
a166 %1.src debug                      :end
if errorlevel 1 goto end
del %1.src
:end
```

To use these batch files, simply enter either

    reline1 *asmfile*

or,

    reline2 *asmfile*

use *asmfile* without a file extension.

3 HITEX HITOP TELEMON 80C167

When using the Hitex telemon 80C167 execution environment, the
following operation remarks exist:

• The following resources are used by the monitor:

00000h	–	079FFh	monitor code
40000h	–	401FFh	monitor vector table
40200h	–	415FFh	monitor data
0FCE0h	–	0FCFFh	register bank
0FA00h	–	0FA3Fh	system stack

You should use the **l166** RESERVE MEMORY locator control to prevent the
locator from locating sections in these regions.

For example:

```
RE(  ME( 00000hTO 079FFh ),
     ME( 40000hTO 401FFh ),
     ME( 40200hTO 415FFh ),
     ME( 0FCE0hTO 0FCFFh ) )
```

`cstart[x orl].asm` uses SSKDEF 0 (256 words) by default and
initializes SP to the top of the system stack (0FBFF). So there is no conflict
with the system stack area of the monitor.

• In the `start.asm` file, the @EVA symbol must be enabled (set to 1).

• The TASKING **ieee166** converter must be used to generate an
   IEEE-695 output file from the absolute (located) output file. The Hitex
   SP166TA filter program is needed to translate this IEEE-695 file into
   Hitex format.

• Bit variables, bitword variables and bit fields are supported by HiTOP,
   but not when using HiTOP with a telemon.

DEBUG ENVIRONMENT

4 PLS FAST-VIEW66

When using the *fast*-view66 debugger, the following operation remarks exist:

- Use the **–g** compiler option to generate debug information for use with *fast*-view66.
- *Fast*-view66 supports all C/C++ language debug information generated by **c166**/**cp166**.
- You can use the absolute output file format (locator output file) for download to the C166/ST10 target hardware.

DEBUG ENVIRONMENT

APPENDIX C

CPU FUNCTIONAL PROBLEMS

TASKING
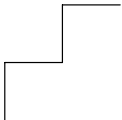
1 INTRODUCTION

Infineon Components and STMicroelectronics regularly publishe 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, the TASKING C166 compiler provides 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 Infineon / STMicroelectronics errata sheets for the CPU step you are using, to verify if you need to use one of these bypasses.

2 CPU FUNCTIONAL PROBLEM BYPASSES

BUS.18 –– JMPR at jump target address

Infineon / STMicroelectronics reference: BUS.18

Use compiler option:

   –BH

Use libraries:

   `lib\[u]ext2p\*.lib`

If a PEC transfer occurs immediately after a JMPR instruction the program counter can have a wrong value. There are many other requirements before this actually happens, among others the JMPR has to be reached by a jump instruction.

CPU.3 –– MOV(B) Rn,[Rm+#data16] as the last instruction in an extend sequence

Infineon reference: CPU.3

Use compiler option:

   –BE

Use libraries:

   `lib\[u]extp\*.lib`

On older C167 derivatives the last instruction in an extend sequence will use a DPP translation instead of the page or segment number supplied with the extend instruction (EXT*xx*). This problem occurs only when the last instruction of this extend instruction uses the addressing mode Rn, [Rm+#data16]. When you use the **–BE** compiler option the compiler will lengthen the extend sequence with one instruction when it generates an instruction using this addressing mode.

CPU PROBLEMS

CPU.11 –– Interrupted multiply

Infineon reference: CPU.11

Use compiler option:

–BU

Use libraries:

```
lib\[u]166p\*.lib
lib\[u]extp\*.lib
lib\[u]ext2p\*.lib
```

This solution should be used where failures occur for interrupts during the MUL and MULU instructions:

- For C166 derivatives, the compiler option **–BU** emits code using run-time library calls for the multiply operations. In these run-time library calls, the operations are protected against interrupts, so that the problem cannot occur.

- For ext and ext2 derivatives, multiply operations are protected inline using ATOMIC instructions. In some cases, an additional NOP might be generated after the multiply instruction. When you want to use the inline protection, you should use both the compiler options **–x**[**i**] and **–BU**.

When using the **–BU** option you should also link libraries in which the divide operations are protected. The libraries in the directories `lib\166p`, `lib\extp` and `lib\ext2p` also have the divide protected against interrupts, but can be used safely to bypass this CPU problem.

• • • • • • • • •

CPU.16 -- MOVB [Rn], mem

Infineon reference: CPU.16

Use compiler option:

–BF

Use libraries:

```
lib\[u]166p\*.lib
lib\[u]extp\*.lib
lib\[u]goldp\*.lib
```

When the MOVB[Rn],mem instruction is executed, where (a) mem specifies a direct 16-bit byte operand address in the internal ROM/Flash memory, and (b) [Rn] points to an even byte address, while the contents of the word which includes the byte addressed by mem is odd, or [Rn] points to an odd byte address, while the contents of the word which includes the bytes addressed by mem is even, the following problem occurs:

1. when [Rn] points to external memory or to the X-Peripheral (XRAM, CAN, etc.) address space, the data value which is written back is always 00h.

2. when [Rn] points to the internal RAM or SFR/ESFR address space, (a) the (correct) data value [mem] is written to [Rn]+1, i.e. to the odd byte address of the selected word in case [Rn] points to an even byte address, (b) the (correct) data value [mem] is written to [Rn]-1, i.e. to the even byte address of the selected word in case [Rn] points to an odd byte address.

Since internal ROM/Flash/OTP data is referred to as 'const' data, the compiler will prevent generating the MOVB [Rn], mem instruction when even 'const' objects are accessed. The compiler is unaware of the exact location of these objects which is determined at locate time.

CPU PROBLEMS

CPU.18 / Problem 7 / CPU.2 -- Interrupted multiply and divide instructions

Infineon reference: CPU.18, Problem 7 and CPU.2

Use compiler option:

–BM

Use libraries:

```
lib\[u]166p\*.lib
lib\[u]extp\*.lib
lib\[u]ext2p\*.lib
```

This solution should be used where failures occur for interrupts during the MUL, MULU, DIV, DIVU, DIVL and DIVLU instructions:

- For C166 derivatives, the compiler option **–BM** emits code using run-time library calls for the multiply and divide operations. In these run-time library calls, the operations are protected against interrupts, so that the problems cannot occur.
- For ext and ext2 derivatives, multiply and divide operations are protected inline using ATOMIC instructions. In some cases, an additional NOP might be generated after the multiply or divide instruction. When you want to use the inline protection, you should use both the compiler options **–x**[**i**] and **–BM**.

–BM is a workaround for many MUL/DIV problems. Besides CPU.18 it fixes problem 7, problem 13, problem 17, CPU.2 and CPU.11.

When using the **–BM** option you should also link libraries in which the multiply and divide operations are protected.

• • • • • • • • •

CPU 1R006 -- CPU hangup with MOV(B) Rn,[Rm+#data16]

Infineon reference: CPU1R006

Use compiler option:

–BO

Use libraries:

```
lib\[u]extp\*.lib
```

The opcode MOV (B) Rn, [Rm+#data16] can cause the CPU to hang. The problem is encountered under the following conditions:

- [Rm+#data16] is used to address the source operand
- [Rm+#data16] points to the program memory
- a hold cycle has to be generated by the ir_ready signal at the beginning of the operand fetch cycle

Since the compiler is unaware of the actual location the source operand [Rm+#data16] refers to, the generation of this addressing mode is completely surpressed.

CPU.21 -- Incorrect result of BFLDL/BFLDH after a write to internal RAM

Infineon / STMicroelectronics reference: CPU.21

Use compiler option:

–BK

Use libraries:

```
lib\[u]extp\*.lib
```

The result of a BFLDL/BFLDH instruction may be incorrect after a write to internal RAM. This only happens under very specific circumstances.

CPU PROBLEMS

CPU_MOVB_JB -- Jump–bit wrongly executed after byte manipulation instructions

Infineon reference: CPU_MOVB_JB

Use compiler option:

–BZmovbjb

Use libraries:

```
lib\[u]extp\*.lib
```

When a JB/JNB/JBC/JNBS on a GPR follows an instruction that performs a byte write operation (MOVB, ADDB/ ADDBC, ANDB, XORB, ORB, NEGB, CPLB, SUBB/SUBCB) on the same GPR but on the byte that is not been used by the JB/JNB/JBC/JNBS (i.e. the byte where the bit used by the jump is not located), the program flow gets corrupted. That means, the jump may be wrongly taken (or not taken). A side effect of this bug is that further program branches may also lead to illegal instruction fetches (fetches are performed from wrong addresses).

Example 1:

```
; Assume Rx.0 is 0 (x any GPR 0..7)
MOVB RxH, any_value       ; Any Byte write instruction on RxH
JB Rx.0, jump_address     ; WILL BE WRONGLY TAKEN!
```

Example 2:

```
; Assume Rx.y is 1 (x any GPR 0..7; y any bit except bit0)
MOVB RxL, any_value       ; Any Byte write instruction on
                          ; RxL for y= 8..15 or RxH for y= 1..7
JNB Rx.y, jump_address    ; WILL BE WRONGLY TAKEN!
```

Example 3:

```
; Assume Rx.0 is 0(x any GPR 0..7)
MOVB RxH, any_value       ; Any Byte write instruction on R0H
JNB Rx.0, jump_address    ; WILL NOT BE TAKEN!
```

Example 4:

```
; Assume Rx.y is 1(x any GPR 0..7; y any bit except bit0)
MOVB RxL, any_value       ; Any Byte write instruction on
                          ; RxL for y= 8..15 or RxH for y= 1..7
JB Rx.y, jump_address     ; WILL NOT BE TAKEN!
```

Note that the bug is only visible when the bit used for the jump evaluation is set (i.e."1") for all the bits Rx.15 to Rx.1, or not-set (i.e."0") for Rx.0.

• • • • • • • • •

Example 5 (BUG NOT VISIBLE):

```
; Assume Rx.0 is 1 (x any GPR 0..7)
MOVB RxH, any_value          ; Any Byte write instruction on RxH
JB Rx.0, jump_address        ; WILL BE CORRECTLY EXECUTED (TAKEN!)
```

Note that the bug exists also when the byte write operation writes into the GPR using indirect addressing mode or memory addressing mode. This situation includes also the use of PECB/DPECB which destination pointer points into a GPR (i.e. data write is performed on a GPR).

Workaround:

1. Include a NOP between any byte write instruction on a GPR and a jump-bit instruction on the same GPR but on a bit belonging to a different byte. This workaround does not cover writing to a GPR through PEC/DPEC.

2. To protect against PEC/DPECB also, include a 'ATOMIC #1' before all jump-bit instructions.

Because the compiler does not expect a GPR modification through a memory or indirect addressing mode, nor through a PEC/DPEC transfer, it uses workaround 1.

CPU PROBLEMS

CR105893 -- Interrupted division corrupted by division in ISR (interrupt service routine)

Infineon reference: CR105893

Use compiler option:

–BZc166sv1div

Use libraries:

`lib\[u]extp\*.lib`

In the first states of a division several internal control signals are set that are used in later states of the division. If a division is interrupted and in the interrupt service routine (ISR) another division is executed, it overrides the old internal values. After the return the interrupted division proceeds with the (probably wrong) internal states of the last division. The affected internal signals are `dividend_sign`, `divisisor_sign` and `mdl_0`. The first two bits represent the operand signs (=Bit 15). `mdl_0` is set if MDL is 0xFFFF.

Workaround:

Do not interrupt divisions, for example by using an ATOMIC sequence.

CR105981 -- JBC and JNBS with op1 a DPRAM operand (bit addressable) do not work

Infineon reference: CR105981

Use compiler option:

–BZc166sv1jbc

Use libraries:

`lib\[u]extp\*.lib`

The DPRAM address (corresponding to op1) is written back with wrong data. This happens even if the jump is not taken.

Note that these instructions work properly for GPR operands and SFR operands.

Workaround:

> Do not use JBC and JNBS instructions, unless the first operand is a GPR.

CR105619 -- "Phantom interrupt" occurs if Software Trap is cancelled

Infineon reference: CR105619

Use compiler option:

–BZc166sv1trap

Use libraries:

```
lib\[u]extp\*.lib
```

The last regularly executed interrupt is injected again if a software trap is canceled and at the same time a real interrupt occurs. A sequence where this problem occurs is the following:

```
BMOV    R13.1,0FD10h.1
TRAP    #010h
```

Due to the previous operation the TRAP is canceled and at the same time a real interrupt occurs. As a result of this, the last previously executed interrupt is injected and then the real interrupt is injected too (if its priority is high enough).

Conditions for canceling a software TRAP are:

- previous instruction changes SP (explicitly)
- previous instruction changes PSW (implicit or explicitly)
- OCDS/hardware triggers are generated on the TRAP instruction

Note that instructions modifying the PSW are almost all the instructions: arithmetic/logical instructions, MOVs,..... For a detailed list of instructions modifying PSW refer to the User Manual.

Workaround:

> Do not cancel a software trap by inserting a NOP before a TRAP instruction.

CPU PROBLEMS

CR107092 -- Extended sequences not properly handled with conditional jumps

Infineon reference: CR107092

Use compiler option:

–BZc166sv1ext

Use libraries:

```
lib\[u]extp\*.lib
```

Affected are the instructions EXTR, EXTP, EXTPR, EXTS, EXTSR and ATOMIC since the responsible code generates the control signals for all these instructions, however, the effects will differ.

Example:

```
                EXTR    #1
                JB      DP1H.6, JMP_TARGET   ; taken jump
                MOV     MDC, #0000Fh
                MOV     MDH, MDL
                CALL    never_reached
  JMP_TARGET:
                MOV     MDC, #0000Fh
                MOV     MDH, MDL
```

In this example the jump is correctly executed and taken. However, the control signal for the extended register sequence is not reset. So, at JMP_TARGET the extend sequence is still effective. This means that the move instruction is extended and instead of writing to the SFR MDC (FF0Eh) the move instruction writes to address F10Eh, an ESFR address.

The bug occurs with taken conditional jumps only, since they are executed as two cycle commands and therefore re-injected in the pipeline. If the jump is not taken or unconditional, the sequence above will work properly, since these jumps are executed as single cycle commands! With "extr #2" in the sequence above, the second move will be affected as well! ATOMIC instructions seem to be a minor issue, since they do not create invalid accesses; in this case the consequence of the bug is that the ATOMIC sequence will be extended to the target instructions also.

Workaround:

Do not jump from extend sequences.

CR108309 –– MDL access immediately after a DIV causes wrong PSW values

Infineon reference: CR108309

Use compiler option:

   –BZinsert_div_mdl

Use libraries:

   A workaround is not default enabled in the libraries. Rebuild the
   libraries manually:

for C files:	use compiler option **–BZinsert_div_mdl**
for `.src` files:	use assembler control `CHECKC166SV1DIVMDL`
for `.asm` files:	use **m166** control `DEF(FIX_EXT1MDL)`

If the MDL register is accessed immediately after a DIV instruction, the
PSW flags are set incorrectly. The problem only appears with DIVs
instructions when they are immediately followed by one instruction that
reads MDL, from type:

```
MOVs mem,reg
ADDs/SUBs/ORs/XORs mem,reg
ADDs/SUBs/ORs/XORs reg,mem
ADDs/SUBs/ORs/XORs reg,mem
CMPs reg.mem
CMPs reg, #data16
```

The V flag can be wrongly calculated for signed divisions: the V flag is
only set if the most significant bit of the result is set (that is, if the result is
negative).

Workaround:

   Insert a NOP instruction after DIV instructions:

```
DIVL R0
NOP
MOV R1, MDL
```

CPU PROBLEMS

CR108342 -- Lost interrupt while executing RETI instruction

Infineon reference: CR108342

Use compiler option:

–BZcpu_reti_int

Use libraries:

no solution in libraries required

The bug occurs when two interrupts are trying to get into the CPU while a RETI instruction is being executed. In this case it can happen that one interrupt is lost (the first one, even if it has a higher priority). Furthermore, the program flow after the ISR can be broken. Only the RETI instruction is affected by this bug. This is because this instruction is specially managed. The instruction following the RETI is internally marked as not interruptable. This means that no interrupt will be served by the CPU between the RETI and its following instruction. This bug is the consequence of an error in how this special treatment is implemented in the logic, specifically in the generation of the "not interruptable" indication.

This workaround marks the instruction following the RETI as not interruptable, (emulating what the hardware was supposed to do).

CR108400 -- Broken program flow after not taken JMPR/JMPA instruction

Infineon reference: CR108400

Use compiler option:

–BZcpu_jmpra_cache

Use libraries:

A workaround is not default enabled in the libraries. Rebuild the libraries manually:

for C files:	use compiler option **–BZcpu_jmpra_cache**
for `.src` files:	use assembler control `CHECKCPUJMPRACACHE`
for `.asm` files:	use **m166** control `DEF(FIX_JMPRA_CACHE)`

This bug can occur in two situations:

1. If the instruction sequentially following a conditional JMPR and/or JMPA is the target instruction of another previously executed JB, JNB, JNBS, JMPR or JMPA, the program flow can be corrupted when the JMPR/JMPA is not taken.

2. If a not-taken JMPR and/or JMPA is inside a loop or a sequence that is executed more than once (caused by a CALL, RET, JMPI, JMPS or TRAP).

The bug occurs because the instruction sequentially following the not-taken jump is fetched from memory but the "identifier" corresponding to this instruction is taken from the jump cache (since this instruction was previously loaded in the jump cache). As a consequence, both instruction an identifier do not match exactly.

CR108904 –– DIV/MUL interrupted by PEC when the previous instruction writes in MDL/MDH

Infineon reference: CR108904

Use compiler option:

–BZinsert_mdl_muldiv

Use libraries:

A workaround is not default enabled in the libraries. Rebuild the libraries manually:

for C files:	use compiler option **–BZinsert_mdl_muldiv**
for `.src` files:	use assembler control CHECKC166SV1MULDIVMDLH
for `.asm` files:	use **m166** control DEF(FIX_EXT1MDLMULDIV)

If the source pointer of PEC/DPEC/EPEC points to the SFR/ESFR area
(PD-bus), the read operation to this SFR/ESFR location is not performed
when the PEC/DPEC/EPEC interrupts a DIV/MUL instruction in its first
execution cycle AND the DIV/MUL instruction follows an instruction
writing into MDL/MDH. In this case, apart from the fact that the read
operation is not performed, the value that is written into the destination
pointer is always FFFFh (default value of the PD-bus).

The instruction sequences affected, are:

```
MOV mdh, Rw    ; or any instruction that writes in
               ; MDH/MDL (1) using any addressing mode
               ; (also indirect addressing and bitaddr)

DIV Rw         ; or DIVL/DIVLU/DIVU or MUL/MULU
```

[1] Writes into ESFR addresses (F00Ch and F00Eh instead of MDH (FE0Ch)
and MDL (FE0Eh)) cause the same problem.

Workaround:

There are two possible workarounds:

1. Insert a NOP instruction (or another instruction not writing into
   MDL/MDH) between an instruction writing into MDL/MDH and a
   DIV/MUL instruction.

2. Do not allow interruption of DIV/MUL by using ATOMIC.

Kfm_BR03 -- Pipeline conflict after CoSTORE

STMicroelectronics reference: Kfm_BR03

Use compiler option:

–BN

Use libraries:

```
lib\[u]extp\*.lib
```

After a CoSTORE instruction with any destiniation (E)SFR, the (E)SFR
cannot be read.

• • • • • • • • •

LONDON.1 –– Breakpoint before JMPI/CAL

Infineon / STMicroelectronics reference: LONDON1

Use compiler option:

–BL

Use libraries:

lib\[u]ext2p*.lib

Description:

JMPI

When the program hits a breakpoint right before a JMPI instruction, the first instruction injected in the pipeline will not be processed by the core. This leads to a deny of all interrupts and OCE injection requests. The problem may also occur when single stepping right before a JMPI instruction.

CALLI

CALLI instruction is not working properly in some cases if it is followed by an injected interrupt. This results in causing a fault in the stack pointer management.

CPU PROBLEMS

LONDON.1751 -- Write to core SFR while DIV[L][U] executes

Infineon / STMicroelectronics reference: LONDON 1751

Use compiler option:

–BA

Use libraries:

```
lib\[u]ext2p\*.lib
```

In the following situation:

```
DIVU R12
ADD  R13, R14
...
MOV  MSW, ....  will destroy the division
...
MOV  R13,MDH
,.
```

Problem 13 -- Interrupted signed division

Infineon reference: Problem 13

Use compiler option:

–BD (**–BM** can also be used)

Use libraries:

```
lib\[u]166p\*.lib
lib\[u]extp\*.lib
lib\[u]ext2p\*.lib
lib\[u]goldp\*.lib
```

Signed divide operations may produce incorrect results when an interrupt
(PEC, standard interrupt or hardware trap) occurs during an execution of
the DIV or DIVL instuction. Note that this bug will not occur for unsigned
divisions. When the **–BD** option is used the compiler will disable
interrupts during a signed division. When the **–BM** option is used all
multiply and divide instructions will be protected against interrupts. This
bypasses several other CPU problems as well.

• • • • • • • • •

Problem 17 -- Interrupted multiply with RETI

Infineon reference: Problem 17

Use compiler option:

–BI

Use libraries:

no solution in libraries required

When a multiply instruction has been interrupted, it may be completed incorrectly after return from interrupt if a higher priority interrupt or hardware trap is generated while the RETI instruction is executed. This problem does not occur with PEC transfers.

In this case the previously mentioned workaround can be used, but at the price of an increased worst case interrupt response time.

To avoid having to use the previous workaround, the problem can be bypassed by an adaption in the interrupt frame code (file `intrpt.c` in the `c` subdirectory of the **examples** directory).

In this file the RETI instruction is preceded by a BFLDH PSW, #0F0h, #0F0H instruction, when the compiler bypass option **–BI** is used. This will cause an interrupted multiplication or division to be correctly completed after RETI before a higher priority interrupt will be acknowledged.

CPU PROBLEMS

Problem S1 -- Byte Write to FLASH EPROM

Infineon reference: Problem S1

Use compiler option:

   –BB

Use libraries:

```
lib\[u]166p\*.lib
```

This problem occurs on older steps of the FLASH EPROM version of the CPU. With the **–BB** option the compiler generates two NOP instructions after each instruction which does a byte write operation. These instructions are: ADDB, ADDCB, ANDB, CPLB, MOVB, NEGB, ORB, SUBB, SUBCB, XORB. The pragma **fix_byte_write** and **nofix_byte_write** can be used to switch this option on the fly in your source code. To reduce the number of NOP instructions to be generated, you can use the disassembler **d166** to detect where erroneous sequences are generated for the CPU.

See the description of the disassembler in the *Utilities* chapter of the *C166/ST10 Assembler, Linker/Locator, Utilities User's Manual* for more information.

ST_BUS.1 –– JMPS followed by PEC transfer

STMicroelectronics reference: ST_BUS.1

Use compiler option:

–BJ

Use libraries:

```
lib\[u]extp\*.lib
```

When a JMPS instruction is followed by a PEC transfer, the generated PEC source address is false. This results in an incorrect PEC transfer.

The compiler prevents the JMPS instruction from interfering with the PEC transfers by inserting an ATOMIC #2 instruction before a JMPS instruction. This bypass option can only be used in combination with the extended instuction set. Further more, all JMPS instructions in the interrupt vector table are replaced by CALLS instructions. The compiler will generate an ADD SP, #04h instruction in the interrupt frame to delete the return address generated by the CALLS instruction from the system stack.

The assembler contains the $CHECKSTBUS1 control to check for this CPU problem.

The instruction to delete the return address from the system stack is part of the interrupt frame and will NOT be generated if **#pragma noframe** was used.

CPU PROBLEMS

APPENDIX D

USER STACK MODEL

TASKING
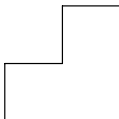
APPENDIX D

1 INTRODUCTION

This appendix describes the special coding methods used in the libraries and C166/ST10 C compiler to support a special stack frame. This appendix describes a user stack model approach, which is used in a special version of the libraries.

If you use the **–P** option of **c166**, the compiler does not emit the regular CALL/RET instructions, when calling a C function, but emits code using a jumping mechanism, specifying the return address on the user stack. The advantage of this approach is that the system stack is not used for the function return address. The price paid for this feature is a run-time execution speed performance penalty. The special libraries needed to support this feature are included in the C and C++ compiler packages.

There are two valid reasons to use this option (and libraries):

*   RTOS

    When using a RTOS kernel, it is often not allowed to use the system stack area (in fact change SP), because this area is reserved for the kernel. Therefore, the **–P** option **must** be used when using RTOS.

*   Heavy recursion

    When the system stack area is getting too small and it is not possible to implement a circular system stack approach (using SOV/SUN exception handlers), the **–P** option can be used. In this case the compiler uses the user stack instead of the system stack. You must link the application with the user stack model libraries.

Using **–P** does not mean that you have to use a RTOS. You can run the application as a standalone application, without any kernel.

The calling convention is explained in more detail in the next chapters.

The push and pop instructions are only allowed during hardware task switches. Nevertheless, with the C compiler option **–Ou**, it is possible to use the user stack instead of the system stack for hardware task switches. See the **–Ou** option in section 4.3 *Detailed Description of the Compiler options* in this manual.

The offset of structure components relative to the structure can be determined from the symbolic debug information, also needed for high level language debugging, generated by the C compiler when you use the command line option **–g**. The syntax for structure symbolic debug information is described in section 3.18 *Structure Type* of the document "C166/ST10 Symbolic Debug Specification".

The conventions for register and data page usage, as well as the calling conventions for functions, are fully documented in chapter 3 *Language Implementation*. Section 3.6 *Function Parameters* of chapter 3, describes when parameters are passed via registers and when they are passed via the user stack.

2 FUNCTION CALL AND RETURN

The next sections describe how function calls and function returns are implemented in the libraries and in the C compiler to support a special stack frame.

2.1 DIRECT INTRA-SEGMENT FUNCTION CALL AND RETURN

A direct intra-segment function call (near function call) is normally performed with a CALLA instruction and returned with a RETN instruction. But the direct intra-segment function call must be performed without using the system stack.

Therefore, the user stack is used to pass the return label to the near function. Then the near function is invoked using an absolute intra-segment jump. At exit, the near function return is implemented using an indirect jump on the contents of the user stack.

The following assembly listing displays the code the C compiler generates for an absolute near function call. The near function called is named _f. Rn is a register used by the C compiler for temporary results.

USER STACK MODEL

```
                                                              min.
                                                       code  state
             .                                         size  times
             .
        mov   Rn, #SOF __RETURN_LABEL                     4     2
        mov   [-R0], Rn                                   2     2
        jmpa  CC_UC, _f                                   4     4
__RETURN_LABEL:
             .                                           --    --
             .                                           10     8
```

The assembly listing described below displays the code the C compiler
generates to return to the caller of the near function.

```
                                                              min.
                                                       code  state
             .                                         size  times
             .
        mov   R2, [R0+]                                   2     2
        jmpi  CC_UC, [R2]                                 2     4
        retv            ; virtual return                  0     0
             .                                           --    --
             .                                            4     6
```

Temporary register R2 is used to pop the return address from the user
stack and to continue program execution at the return label via a indirect
jump on the contents of R2. The user stack pointer is updated by the
called function before it returns (see [R0+]). This is not the regular
method to handle the user stack pointer in a C function, but this saves one
instruction. Register R2 can be used, because it is always free for use at
function return. No parameters are returned via register R2.

2.2 INDIRECT INTRA-SEGMENT FUNCTION CALL AND RETURN

An indirect intra-segment function call (indirect near function call) must
also be performed without using the system stack. The user stack is used
to pass the return label to the near function. The (offset) address of the
near function is determined at run-time. At exit, the near function returns
the same way as described above.

The following assembly listing displays the code the C compiler generates
for an indirect near function call. The near function called indirectly is in
the function pointer array named _fp. Rx contains the index value. Rn is a
register used by the C compiler for temporary results.

• • • • • • • • •

```
                                                         min.
                                               code    state
        .                                      size    times
        .
        mov   Rn, #SOF __RETURN_LABEL             4       2
        mov   [-R0], Rn                           2       2
        mov   Rn, [Rx+#_fp]                       4       4
        jmpi  CC_UC, [Rn]                         2       4
__RETURN_LABEL:
        .                                        --      --
        .                                        12      12
```

It is obvious that the code, needed to return from a near function, is
always the same, because the function does not know whether it is called
directly or indirectly. See the previous section for the code the C compiler
generates to return from a near function.

2.3   INDIRECT INTER-SEGMENT JUMPS USING THE EXTENDED INSTRUCTION SET

Because the extended instruction set is available (e.g. C167) the system
stack can be used for indirect inter-segment jumps. Because with the
extended instruction ATOMIC the standard PEC interrupts and class A
hardware trap can be disabled for a specified number of instructions.

To perform an indirect inter-segment jump the segment number and
segment offset are pushed on the system stack and a RETS instruction is
executed. Then the execution resumes at the inter-segment address
pushed on the system stack. To avoid that these instructions are
interrupted they are protected with an ATOMIC instruction.

The following assembly listing shows the code for an indirect
inter-segment jump using the ATOMIC instruction. Rseg and Rsof contain
the inter-segment address to jump to.

```
                  ; code can be read as :
                  ; mov CSP, Rseg
                  ; mov IP, Rsof
        atomic #3         ; protect against interrupts
        push  Rseg        ; (SP) <- (SP) - 2 ; ((SP)) <- Rseg
        push  Rsof        ; (SP) <- (SP) - 2 ; ((SP)) <- Rsof
        rets              ; (IP) <- ((SP))   ; (SP)   <- (SP) + 2
                          ; (CSP)<- ((SP))   ; (SP)   <- (SP) + 2
```

USER STACK MODEL

2.4   DIRECT INTER-SEGMENT FUNCTION CALL AND RETURN

A direct inter-segment function call (far function call) is normally performed with a CALLS instruction and returned with a RETS instruction, but now the system stack may not be used to store the function return address.

A direct inter-segment function can be invoked using a JMPS instruction, but the called function does not know where to return to on exit. Therefore, before a direct inter-segment jump can be performed to the far function, the segment number and segment offset of the return label must be stored on the user stack. The far function being invoked returns to its caller by getting the return label from the user stack and then performing an indirect inter-segment jump to the return label, as described in the previous section.

The next assembly listing displays the code the C compiler generates for a far function call when extended instructions are available.

The far function called is named **_f**. **Rsof** and **Rseg** are registers used by the C compiler for temporary results.

```
                                                        min.
                                                 code   state
              .                                  size   times
              .
        mov   Rsof, #SOF __RETURN_LABEL            4      2
        mov   [-R0], Rsof                          2      2
        mov   Rseg, #SEG __RETURN_LABEL            4      2
        mov   [-R0], Rseg                          2      2
        jmps  SEG _f, SOF _f                       4      4
__RETURN_LABEL:                                   --     --
              .                                   16     12
              .
```

The next assembly listing displays the code the C compiler generates for a far function to return to its caller.

```
                                                                     min.
                                                         code       state
    .                                                    size       times
    .
    mov   Rseg, [R0+]                                       2           2
    mov   Rsof, [R0+]                                       2           2
    atomic #3                                               2           2
    push  Rseg                                              2           2
    push  Rsof                                              2           2
    rets                                                    2           4
    .                                                      --          --
    .                                                      12          14
```

2.5 INDIRECT INTER-SEGMENT FUNCTION CALL AND RETURN

An indirect inter-segment function call (indirect far function call) is normally performed with a run-time library function, and the far function called indirect returns with a RETS instruction. The segment number and segment offset are passed to this run-time library function to perform the inter-segment call, but it uses the system stack which is not allowed in this implementation of the library.

Also now the segment number and segment offset of the return label must be stored on the user stack before an indirect inter-segment jump can be performed to the far function. The far function being invoked returns to its caller by getting the return label from the user stack and then performing an indirect inter-segment jump to the return label. The far function being invoked is determined run-time. So, an indirect inter-segment jump is needed. When segment number and segment offset of the far function being called is determined run-time, the same mechanism as described in section 2.4, can be used again to make the inter-segment jump.

The next assembly listing displays the code the C compiler generates for an indirect far function call when extended instructions are available.

The far function called indirectly is in the function pointer array named _fp. Rx contains the index value. Rseg and Rsof are registers used by the C compiler for temporary results.

USER STACK MODEL

```
                                                                    min.
                                                         code    state
                        .                                size    times
                        .
        mov   Rsof, #SOF __RETURN_LABEL                    4        2
        mov   [-R0], Rsof                                  2        2
        mov   Rseg, #SEG __RETURN_LABEL                    4        2
        mov   [-R0], Rseg                                  2        2
        mov   Rsof, [Rx+#_fp ]                             4        4
        mov   Rseg, [Rx+#_fp+02H]                          4        4
        atomic #3                                          2        2
        push  Rseg                                         2        2
        push  Rsof                                         2        2
        rets                                               2        4
__RETURN_LABEL:                                           --       --
                        .                                 28       26
                        .
```

It is obvious that the code, needed to return from a far function is always the same, because the function does not know whether it is called directly or indirectly. See section 2.4 for the code the C compiler generates to return from a far function when extended instructions are available.

3 MIXING USER STACK AND NON-USER STACK FUNCTIONS

With the _usm and _nousm function qualifiers, the compiler is instructed to generate a user stack model calling convention regardless of the usage of the compiler option option –P (see section 3.2.2, *User Stack Model*).

USER STACK MODEL

INDEX

INDEX

Symbols

INDEX

A

INDEX

D

INDEX

E

INDEX

M

INDEX

INDEX

INDEX

INDEX

X

INDEX