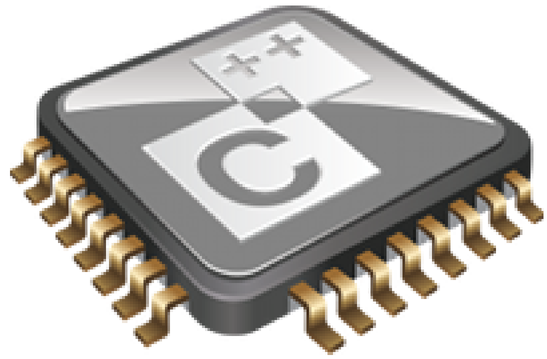
Embedded Studio Reference Manual

Version: 3.10i

Contents



Introduction

This guide is divided into a number of sections:

Introduction

Covers installing SEGGER Embedded Studio on your machine and verifying that it operates correctly, followed by a brief guide to the operation of the SEGGER Embedded Studio integrated development environment, debugger, and other software supplied in the product.

SEGGER Embedded Studio User Guide

Contains information on how to use the SEGGER Embedded Studio development environment to manage your projects, build, and debug your applications.

C Library User Guide

Contains documentation for the functions in the standard C library supplied in SEGGER Embedded Studio.

ARM target support

Contains a description of system files used for startup and debugging of ARM applications.

ide_target_interfaces

Contains a description of the support for programming ARM microcontrollers.

What is SEGGER Embedded Studio?

SEGGER Embedded Studio is a complete C/C++ development system for ARM and Cortex, microcontrollers and microprocessors that runs on Windows, Mac OS and Linux.

C/C++ Compiler

SEGGER Embedded Studio comes with pre-built versions of both GCC and Clang/LLVM C and C++ compilers and assemblers. The GNU linker and librarian are also supplied to enable you to immediately begin developing applications for ARM.

SEGGER Embedded Studio C Library

SEGGER Embedded Studio has its own royalty-free ANSI and ISO C compliant C library that has been specifically designed for use within embedded systems.

SEGGER Embedded Studio C++ Library

SEGGER Embedded Studio supplies a C++ library that implements STL containers, exceptions and RTTI.

SEGGER Embedded Studio IDE

SEGGER Embedded Studio is a streamlined integrated development environment for building, testing, and deploying your applications. SEGGER Embedded Studio provides:

Source Code Editor:A powerful source code editor with multi-level undo and redo, makes editing your code a breeze.
Project System:A complete project system organizes your source code and build rules.
Build System:With a single key press you can build all your applications in a solution, ready for them to be loaded onto a target microcontroller.
Debugger and Flash Programming:You can download your programs directly into Flash and debug them seamlessly from within the IDE using a wide range of target interfaces.
Help system:The built-in help system provides context-sensitive help and a complete reference to the SEGGER Embedded Studio IDE and tools.
Core Simulator:As well as providing cross-compilation technology, SEGGER Embedded Studio provides a PC-based fully functional simulation of the target microcontroller core so you can debug parts of your application without waiting for hardware.

SEGGER Embedded Studio Tools

SEGGER Embedded Studio supplies command line tools that enable you to build your application on the command line using the same project file that the IDE uses.

What we don't tell you

This documentation does not attempt to teach the C or assembly language programming; rather, you should seek out one of the many introductory texts available. And similarly the documentation doesn't cover the ARM architecture or microcontroller application development in any great depth.

We also assume that you're fairly familiar with the operating system of the host computer being used.

C programming guides

These are must-have books for any C programmer:

Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.
The original C bible, updated to cover the essentials of ANSI C (1990 version).
Harbison, S.P. and Steele, G.L., *C: A Reference Manual* (second edition, 1987). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-109802-0.
A nice reference guide to C, including a useful amount of information on ANSI C. Co-authored by Guy Steele, a noted language expert.

ANSI C reference

If you're serious about C programming, you may want to have the ISO standard on hand:

ISO/IEC 9899:1990, C Standard and ISO/IEC 9899:1999, C Standard. The standard is available from your national standards body or directly from ISO at **http://www.iso.ch/**.

ARM microcontrollers

For ARM technical reference manuals, specifications, user guides and white papers, go to:

http://www.arm.com/Documentation.

GNU compiler collection

For the latest GCC documentation go to:

http://gcc.gnu.org/.

LLVM/Clang

For the latest LLVM/Clang documentation to to:

http://www.llvm.org

Getting Started

You will need to install a CPU support package:

Choose **Tools** > **Package Manager**
Choose the CPU support packages you wish to install and complete the dialog.

You will need to create a project:

Choose **File** > **New Project**
Select the appropriate Executable project type
Specify a location for the project
Complete the dialog selecting the appropriate **Target Processor** value

You will need to build the project:

Choose **Build | Build 'Project'**

To debug on the simulator

Choose **Project | Edit Options** to show the project options dialog
In the **Search Options** type in **Simulator**
Choose **Simulator** for the **Target Connection** option

To debug on hardware

Choose **Project | Edit Options** to show the project options dialog
In the **Search Options** type in **J-Link**
Choose **J-Link** for the **Target Connection** option

To start debugging

Choose **Debug | Go**

The debugger will stop the program at the main, you can now debug the application.

Text conventions

Menus and user interface elements

When this document refers to any user interface element, it will do so in **bold font**. For instance, you will often see reference to the **Project Explorer**, which is taken to mean the project explorer window. Similarly, you'll see references to the **Standard** toolbar which is positioned at the top of the SEGGER Embedded Studio window, just below the menu bar on Windows and Linux.

When you are directed to select an item from a menu in SEGGER Embedded Studio, we use the form *menu-name* > *item-name*. For instance, **File** > **Save** means that you need to click the **File** menu in the menu bar and then select the **Save** item. This form extends to items in sub-menus, so **File** > **Open With Binary Editor** has the obvious meaning.

Keyboard accelerators

Frequently-used commands are assigned keyboard *accelerators* to speed up common tasks. SEGGER Embedded Studio uses standard Windows and Mac OS keyboard accelerators wherever possible.

Windows and Linux have three key modifiers which are **Ctrl**, **Alt**, and **Shift**. For instance, **Ctrl+Alt+P** means that you should hold down the **Ctrl** and **Alt** buttons whilst pressing the **P** key; and **Shift+F5** means that you should hold down the **Shift** key whilst pressing **F5**.

Mac OS has four key modifiers which are (command), (option), (control), and (shift). Generally there is a one-to-one correspondence between the Windows modifiers and the Mac OS modifiers: **Ctrl** is , **Alt** is , and **Shift** is . SEGGER Embedded Studio on Mac OS has its own set of unique key sequences using (control) that have no direct Windows equivalent.

SEGGER Embedded Studio on Windows and Linux also uses *key chords* to expand the set of accelerators. Key chords are key sequences composed of two or more key presses. For instance, the key chord **Ctrl+T, D** means that you should type **Ctrl+T** followed by **D**; and **Ctrl+K, Ctrl+Z** means that you should type **Ctrl+T** followed by **Ctrl+Z**. Mac OS does not support accelerator key chords.

Code examples and human interaction

Throughout the documentation, text printed **in this typeface** represents verbatim communication with the computer: for example, pieces of C text, commands to the operating system, or responses from the computer. In examples, text printed *in this typeface* is not to be used verbatim: it represents a class of items, one of which should be used. For example, this is the format of one kind of compilation command:

hcl *source-file*

This means that the command consists of:

The word **hcl**, typed exactly like that.
A *source-file*: not the text **source-file**, but an item of the *source-file* class, for example **myprog.c**.

Whenever commands to and responses from the computer are mixed in the same example, the commands (i.e. the items which you enter) will be presented `in this typeface`. For example, here is a dialog with the computer using the format of the compilation command given above:

```
c:\code\examples>hcl -v myprog.c
```

The user types the text **hcl -v myprog.c** and then presses the enter key (which is assumed and is not shown); the computer responds with the rest.

Release notes

Version 3.10i

Build

Fixed C++ signbit implementation.

Fixed build not building newly imported files.

Debug

Fixed misc bugs in watch window.

Fixed crash when setting register allocated variables that are less than 4 bytes in length.

Fixed Cortex-M simulator handling of word aligned stack pointers on exception return.

Fixed crash using debug stop during download.

Updated shipped J-Link software to V6.12a.

IDE

Fixed pasting of file into a project explorer folder that is already contained within that folder.

Fixed file overwrite warning dialog not giving the option to cancel.

Fixed crash when closing all editor windows, right clicking and then moving the mouse with the empty dock window.

Fixed editor search not clearing highlights when search string is cleared.

Version 3.10h

Build

Removed broken linker variant project property.

Debug

Fixed watch window not storing the filename and linenumber context in which to evaluate the expression.

Updated shipped J-Link software to V6.10n.

IDE

Fixed the window group context menu sometimes being displayed higher than the mouse pointer.

Search box now focused when package manager is opened.
Fixed project importer not loading shipped jlink.dll when no other is available.

Version 3.10g

Debug

Fixed SWO tracing.

Version 3.10f

Build

Fixed crash caused by clearing build log whilst building.
Fixed ld executable being loaded into an editor on memory segment overflow linker errors (Linux and macOS only).
Fixed setting strict-aligned clang compiler option for v4t, v5te, v6m and v8m.baseline architectures.

IDE

Fixed use of proportional fonts in code editor.
Fixed size of .emProject, .emPackage and .emArchive file icons (Linux only).
Added .svd file type detection to "Register Definition File" property.

Version 3.10e

IDE

Ctrl+C and Ctrl+A now work in project property dialog's description field.
Fixed drag not working in project explorer on new files or folders until project has been reloaded.

Version 3.10d

Debug

Fixed memory window starting in auto size mode when a fixed size is specified.

Fixed crash when opening memory window.

Version 3.10c

Debug

Fixed memory window always evaluating address expression when auto size is selected.

IDE

Fixed build summary not being fully visible when build log is larger than window.
Fixed directory chooser on IAR/Keil project import dialog.
Fixed crash when entering address in disassembly window that is not within the current visible address range.
Fixed text editor slow down when rendering very long lines.
Find dialog auto complete is now case sensitive.
Enter key now behaves the same as return key in find dialog.
Fixed crash when dragging a project folder onto one of its own sub folders.
Register window bitfield entries now use monospace font.

Version 3.10b

Debug

Fixed crash using debug restart with the simulator target.
Updated shipped J-Link software to V6.00i.
Add "Ignore .debug_frame Section" debugging property.
Fixed crash if memory write fails during download on OS X.
Fixed crash when connecting to J-Link target interface when no J-Link is attached.
Fixed disassembly of ldr literal instructions.
Improved debug support of clang generated code.

IDE

Fixed goto definition and find references on files with paths containing UNICODE characters.
File and project history now shared between all major releases.
Fixed save prompt dialog not being displayed when modified file is closed.
Fixed close solution not stopping when the saving of a modified file has failed.
Improved appearance of debugger tooltips.

Version 3.10a

Debug

Added "Auto Refresh" to the context menu of the execution count window.
Fixed set breakpoint on variable from text editor.
Fixed modifying breakpoint properties.
Fixed crash when pressing page down on last page of the disassembly in the disassembly window.

IDE

Parallel building of files in projects now shows a progress bar and ETA, both of these can be disabled using new environment options.
Fixed Microsoft IME support in code editor.
Removed project property "Build Dependents in Parallel", replaced it with "Project Can Build In Parallel".
Fixed saving of files from code editor using Shift-JIS encoding.
Improved appearance of build log's memory usage summary when there are many memory segments.
Fixed code completion on files with paths containing UNICODE characters.

Version 3.10

Build

Added "Keep Linker Script File" project property.
Added "Tool Chain Directory" project property.
Added "Externally Built Library" to the "Project Type" enumeration.
Added "Section Placement Segments" project property.
Added "Post-Link Output File" project property.
Added "C Compile Command", "C++ Compile Command", "Assemble Command", "Link Command" and "Archive Command" project properties to enable user defined build steps.
Added capability to build "Externally Built Executable" and "Externally Built Library" project types using command lines in project properties.
Supplied ascii only version of ctype functions in ctype_no_wchar.c in the $(StudioDir)/source directory.
Supplied non threaded version errno in errno_no_thread.c in the $(StudioDir)/source directory.
Fixed *0.0 >= 0.0* and *0.0 <= 0.0* not returning true on some architectures.
Fixed crash when building libraries using indirect files with no files to archive.
Opening IAR EWARM/Keil MDK project files will now create internal and external build configurations.
The external build configuration will use the IAR/Keil tools. The internal build configuration will use the internal tools.
Fixed setting errno to EDOM with invalid arguments to acos(f), asin(f) and fmod(f).
Fixed setting errno to ERANGE when overflow occurs with ldexp(f).
Updated the GCC/BINUTILS tools build to use GCC ARM Embedded 5-2016-q2-update source release.

Link will now fail with an error if a section has been placed at the end of a segment using the *place_from_segment_start* attribute and the preceding section overlaps it.

Debug

Removed "J-Link DLL File" project property.
Added "Use Built-in RTT support" project property to enable RTT to be used on all devices.
Add "Ignore .debug_aranges Section" debugging property.
Fixed crash while using memory window when not connected to target.
Added "Load ELF Sections" loader project property.
Added "Stop On Memory Error" simulator project property.
Removed "Debugger Initial Breakpoint" environment options.
Can now optionally specify the main load file to download using the "Load File" project property.
Added "Go To Disassembly" to code editor context menu when debugging.
Added "Export As Text" to variable display windows.
Variable display windows now display char * as null terminated strings by default.
Variable display windows now carry out numerical sort when sorting on size.
Fixed watch window not moving blank entry to end when using add to watch from editor.
Adding simulator project property "Stop On Memory Error".
Enabling of exception breakpoints is preserved in session file.

IDE

Project properties dialog now defaults to the active build confguration when opened.
Fixed crash when dragging a project folder onto itself.
Fixed display of string properties that contain line feeds.
The "Project Type" property can now be set on a per configuration basis.
Fixed reload project not working correctly when the project has been opened from the Recent Projects window.
Project properties editor dialog is now modal.
Memory window address field now expands to fill available space.
Fixed blank filenames in **Open File From Solution** dialog.
Fixed crash when right clicking in empty area of build configuration dialog.
Fixed crash when changing configuration with a property editor focused in properties dialog.
Changed the way modified and inherited properties are shown in the properties dialog/window.
Code editor no longer displays file modified warning if file has been externally deleted.
Properties dialog, removed "All" group, deselecting the group/page will show all properties.
Properties dialog, changed the graphic to indicate that a project property has been modified or is set in another node or configuration.
Properties dialog, removed the build macros and added a means to display the set of macros on individual property editors.
Project explorer, added "Exclude From Build" option to folders.

Add support for ARMv8-M Mainline and ARMv8-M Baseline architecture.

Properties dialog, added option to show modified properties only.

Fixed drag and drop in project explorer only working on a new folder after project has been reloaded.

Project explorer removed automatic usage of memory map, section placement and linker script files in the build.

Added "Use This Memory Map", "Use This Section Placement" and "Use This Linker Script File" to the appropriate files in the project explorer context menu.

Files of type Memory Map, Section Placement and Linker Script that are added to the project now prompt to update the appropriate project property.

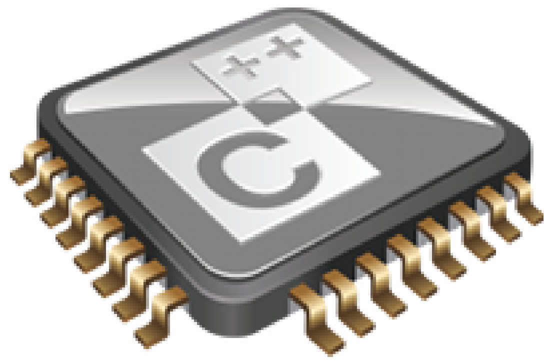
Added "Use Manual Linker Script" and "Linker Script File" project properties.

Removed Calcpad, Debug Immediate, Downloads, Properties, Script Console and Terminal Emulator windows.

Reorganised menu entries, all window show actions are avaiable on the View menu.

File path property editor now applies change when enter key is pressed.

The **Application Monospace Font** property editor will now only allow monospace fonts to be selected.



SEGGER Embedded Studio User Guide

This is the user guide for the SEGGER Embedded Studio integrated development environment (IDE). The SEGGER Embedded Studio IDE consists of:

- a project system to organize your source files
- a build system to build your applications
- programmer aids to navigate and work effectively
- a target programmer to download applications into RAM or flash
- a debugger to pinpoint bugs

SEGGER Embedded Studio standard layout

SEGGER Embedded Studio's main window is divided into the following areas:

Title bar:Displays the name of the current solution.
Menu bar:Menus for editing, building, and debugging your program.
Toolbars:Frequently used actions are quickly accessible on toolbars below the menu bar.
Editing area:A tabbed view of any open editor windows and the HTML viewer.
Docked windows:SEGGER Embedded Studio has many windows that dock to the left, right, or below the editing area. You can configure which windows will be visible, and their placement, when editing and debugging.
Status bar At the bottom of the main window, the status bar contains useful information about the current editor, build status, and debugging environment.

Menu bar

The menu bar contains menus for editing, building, and debugging your program. You can navigate menus using the keyboard or the mouse.

Navigating menus using the mouse

To navigate menus using the mouse:

1. Click a menu title in the menu bar to show the related menu.
2. Click the desired command in the menu to execute that command.

or

1. Click and hold the mouse on a menu title in the menu bar to show the related menu.
2. Drag the mouse to the desired command in the menu.
3. Release the mouse while it is over the command to execute that command.

Navigating menus with the keyboard

To navigate menus using the keyboard:

1. Tap the **Alt** key activate the menu bar.
2. Tap **Return** to display the menu.
3. Use the **Left** and **Right** keys to select the required menu.
4. Use the **Up** or **Down** key to select the required command or submenu.
5. Press **Enter** to execute the selected command.
6. Press **Alt** or **Esc** at any time to cancel menu selection.

After you press the **Alt** key once, each menu on the menu bar has one letter underlinedits shortcut key. So, to activate a menu using the keyboard:

While holding down the **Alt** key, type the desired menu's shortcut key.

After the menu appears, you can navigate it using the cursor keys:

Use **Up** and **Down** to move up and down the list of menu items.
Use **Esc** to cancel a menu.
Use **Right** or **Enter** to open a submenu.
Use **Left** or **Esc** to close a submenu and return to the parent menu.
Type the underlined letter in a command's name to execute that command.

Title bar

The first item shown in the title bar is SEGGER Embedded Studio's name. Because SEGGER Embedded Studio can be used to target different processors, the name of the target processor family is also shown, to help you distinguish between instances of SEGGER Embedded Studio when debugging multi-processor or multi-core systems.

The filename of the active editor follows SEGGER Embedded Studio's name; you can configure the presentation of this filename as described below.

After the filename, the title bar displays status information on SEGGER Embedded Studio's state:

[building] SEGGER Embedded Studio is building a solution, building a project, or compiling a file.
[run] An application is running under control of SEGGER Embedded Studio's debugger.
[break] The debugger is stopped at a breakpoint.
[autostep] The debugger is single stepping the application without user interaction (*autostepping*).

Status bar

At the bottom of the window, the status bar contains useful information about the current editor, build status, and debugging environment. The status bar is divided into two regions: one contains a set of fixed panels and the other is used for messages.

The message area

The leftmost part of the status bar is a message area used for things such as status tips, progress information, warnings, errors, and other notifications.

Status bar panels

You can show or hide the following panels on the status bar:

Panel	Description
Target device status	Displays the connected target interface. When connected, this panel contains the selected target interface's name and, if applicable, the processor to which the target interface is connected. The LED icon flashes green when a program is running, is solid red when stopped at a breakpoint, and is yellow when connected to a target but not running a program. Double-clicking this panel displays the **Targets** pane, and right-clicking it invokes the **Target** shortcut menu.
Cycle count panel	Displays the number of processor cycles used by the executing program. This panel is only visible if the connected target supports performance counters that can report the total number of cycles executed. Double-clicking this panel resets the cycle counter to zero, and right-clicking it brings up the **Cycle Count** shortcut menu.
Insert/overwrite status	Indicates whether the current editor is in insert or overwrite mode. In overwrite mode, the panel displays "OVR"; in insert mode, the panel displays "INS".
Read-only status	Indicates whether the editor is in read-only mode. If the editor is editing a read-only file or is in read-only mode, the panel display "R/O"; if the editor is in read-write mode, the panel displays "R/W".
Build status	Indicates the success or failure of the last build. If the last build completed without errors or warnings, the build status pane contains **Built OK**; otherwise, it contains the number of errors and warnings reported. If there were errors, double-clicking this panel displays the **Build Log** in the **Output** pane.

Caret position	Indicates the insertion position position in the editor window. For text files, the **caret position** pane displays the line number and column number of the insertion point in the active window; when editing binary files, it displays the address being edited.
Time panel	Displays the current time.

Configuring the status bar panels

To configure which panels are shown on the status bar:

Choose **View** > **Status Bar**.
From the status bar menu, select the panels to display and deselect the ones you want hidden.

or

Right-click the status bar.
From the status bar menu, select the panels to display and deselect the ones you want to hide.

To show or hide the status bar:

Choose **View** > **Status Bar**.
From the status bar menu, select or deselect the **Status Bar** item.

You can choose to hide or display the *size grip* when SEGGER Embedded Studio's main window is not maximized. (The size grip is never shown in full-screen mode or when maximized.)

To show or hide the size grip

Choose **View** > **Status Bar**.
From the status bar menu, select or deselect the **Size Grip** item.

Editing workspace

The main area of SEGGER Embedded Studio is the editing workspace. It contains any files being edited, the on-line help system's HTML browser, and the Dashboard.

Docking windows

SEGGER Embedded Studio has a flexible docking system you can use to position windows as you like them. You can dock windows in the SEGGER Embedded Studio window or in the four *head-up display* windows. SEGGER Embedded Studio will remember the position of the windows when you leave the IDE and will restore them when you return.

Window groups

You can organize SEGGER Embedded Studio windows into *window groups*. A window group has multiple windows docked in it, only one of which is *active* at a time. The window group displays the active window's title for each of the windows docked in the group.

Clicking on the window icons in the window group's header changes the active window. Hovering over a docked window's icon in the header will display that window's title in a *tooltip*.

To dock a window to a different window group:

Press and hold the left mouse button over the title of the window you wish to move.
As you start dragging, all window groups, including hidden window groups, become visible.
Drag the window over the window group to dock in.
Release the mouse button.

Holding **Ctrl** when moving the window will prevent the window from being docked. If you do not dock a window on a window group, the window will float in a new window group.

Perspectives

SEGGER Embedded Studio remembers the dock position and visibility of each window in each *perspective*. The most common use for this is to lay your windows out in the **Standard** perspective, which is the perspective used when you are editing and not debugging. When SEGGER Embedded Studio starts to debug a program, it switches to the **Debug** perspective. You can now lay out your windows in this perspective and SEGGER Embedded Studio will remember how you laid them them out. When you stop debugging, SEGGER Embedded Studio will revert to the **Standard** perspective and that window layout for editing; when you return to **Debug** perspective on the next debug session, the windows will be restored to how you laid them out in that for debugging.

SEGGER Embedded Studio remembers the layout of windows, in all perspectives, such that they can be restored when you run SEGGER Embedded Studio again. However, you may wish to revert back to the standard docking positions; to do this:

Dashboard

When SEGGER Embedded Studio starts, it presents the **Dashboard**, a collection of panels that provide useful information, one-click loading of recent projects, and at-a-glance summaries of activity relevant to you.

Tasks

The **Tasks** panel indicates tasks you need to carry out before SEGGER Embedded Studio is fully functionalfor instance, whether you need to activate SEGGER Embedded Studio, install packages, and so on.

Updates

The **Updates** panel indicates whether any packages you have installed are now out of date because a newer version is available. You can install each new package individually by clicking the **Install** button under each notification, or install all packages by clicking the **Install all updates** link at the bottom of the panel.

Projects

The **Projects** panel contains links to projects you have worked on recently. You can load a project by clicking the appropriate link, or clear the project history by clicking the **Clear List** button. To manage the contents of the list, click the **Manage Projects** link and edit the list of projects in the **Recent Projects** window.

News

The **News** panel summarizes the activity of any RSS and Atom feeds you have subscribed to. Clicking a link will display the published article in an external web browser. You can manage your feed subscriptions to by clicking the **Manage Feeds** link at the end of the **News** panel and *pinning* the feeds in the **Favorites** windowyou are only subscribed to the pinned feeds.

Links

The **Links** panel is a handy set of links to your favorite websites. If you pin a link in the **Favorites** window, it appears in the **Links** panel.

SEGGER Embedded Studio help and assistance

SEGGER Embedded Studio provides context-sensitive help in increasing detail:

Tooltips
When you position the pointer over a button and keep it still, a small window displays a brief description of the button and its keyboard shortcut, if it has one.

Status tips
In addition to tooltips, SEGGER Embedded Studio provides a longer description in the status bar when you hover over a button or menu item.

Online manual
SEGGER Embedded Studio has links from all windows to the online help system.

The browser

Documentation pages are shown in the **Browser**.

Help using SEGGER Embedded Studio

SEGGER Embedded Studio provides an extensive, HTML-based help system that is available at all times.

To view the help text for a particular window or other user-interface element:

Click to select the item with which you want assistance.
Choose **Help** > **Help** or press **F1**.

Help within the text editor

The text editor is linked to the help system in a special way. If you place the insertion point within a word and press **F1**, the help-system page most likely to be useful is displayed in the HTML browser. This a great way to quickly find the help text for functions provided in the library.

Browsing the documentation

The **Contents** window lists all the topics in the SEGGER Embedded Studio documentation and gives a way to search through them.

The highlighted entry indicates the current help topic. When you click a topic, the corresponding page appears in the **Browser** window.

The **Next Topic** and **Previous Topic** items in the **Help** menu, or the buttons on the **Contents** window toolbar, help navigate through topics.

To search the online documentation, type a search phrase into the **Search** box on the **Contents** window toolbar.

To search the online documentation:

Choose **Help** > **Contents** or press **Ctrl+Alt+F1**.
Enter your search phrase in the **Search** box and press **Enter** (or **Return** on Macs).

The search commences and the table of contents is replaced by links to pages matching your query, listed in order of relevance. To clear the search and return to the table of contents, click the clear icon in the **Search** box.

Creating and managing projects

A SEGGER Embedded Studio *project* is a container for everything required to build your applications. It contains all the assorted resources and maintains the relationships between them.

A project is a convenient place to find every file and piece of information associated with your work. You place projects into a *solution*, which can contain one or more projects.

This chapter introduces the various parts of a project, shows how to create projects, and describes how to organize the contents of a project. It describes how to use the **Project Explorer** and **Project Manager** for project-management tasks.

Solutions and projects

To develop a product using SEGGER Embedded Studio, you must understand the concepts of *projects* and *solutions*.

A *project* contains and organizes everything you need to create a single application or a library.

A *solution* is a collection of projects and configurations.

Organizing your projects into a solution allows you to build all the projects in a solution with a single keystroke, and to load them onto the target ready for debugging.

In your SEGGER Embedded Studio project, you

- organize build-system inputs for building a product.
- add information about items in the project, and their relationships, to assist you in the development process.

Projects in a solution can reside in the same or different directories. Project directories are always relative to the directory of the solution file, which enables you to more-easily move or share project-file hierarchies.

The **Project Explorer** organizes your projects and files, and provides quick access to the commands that operate on them. A toolbar at the top of the window offers quick access to commonly used commands.

Solutions

When you have created a solution, it is stored in a project file. Project files are text files, with the file extension **emProject**, that contain an XML description of your project. See **Project file format** for a description of the project-file format.

Projects

The projects you create within a solution have a *project type* SEGGER Embedded Studio uses to determine how to build the project. The project type is selected when you use the **New Project** dialog. The available project types depend on the SEGGER Embedded Studio variant you are using, but the following are present in most SEGGER Embedded Studio variants:

- *Executable*: a program that can be loaded and executed.
- *Externally Built Executable*: an executable that is not built by the SEGGER Embedded Studio internal build process.
- *Library*: a group of object files collected into a single file (sometimes called an *archive*).
- *Externally Built Library*: a library that is not built by the SEGGER Embedded Studio internal build process.
- *Object File*: the result of a single compilation.

Staging: a project that will apply a user-defined command to each file in a project.
Combining: a project that can be used to apply a user-defined command when any files in a project have changed.

Project options and configurations

Project options are attached to project nodes. They are usually used in the build process, for example, to define C preprocessor symbols. You can assign different values to the same project option, based on a configuration: for example, you can assign one value to a C preprocessor symbol for release build and a different value for a debug build.

Folders and Dynamic Folders

Projects can contain *folders,* which are used to group related files. Automated grouping uses the files' extensions to, for example, put all .c files in one folder, etc. Grouping also can be done manually by explicitly creating a file within a folder. Note that these project folders do not map onto directories in the file system, they are used solely to structure the display of content shown in the **Project Explorer**.

Projects can also contain *dynamic folders* which will can show the directories and files contained in the file system in the project explorer. You can specify if the dynamic folder is recursive and use wildcards to include and exclude files.

Source files

Source files are all the files used to build a product. These include source code files and also section-placement files, memory-map files, and script files. All the source files you use for a particular product, or for a suite of related products, are managed in a SEGGER Embedded Studio project. A project can also contain files that are not directly used by SEGGER Embedded Studio to build a product but contain information you use during development, such as documentation. You edit source files during development using SEGGER Embedded Studio's built-in text editor, and you organize files into a target (described next) to define the build-system inputs for creating the product.

The source files of your project can be placed in folders or directly in the project. Ideally, the paths to files placed in a project should be relative to the project directory, but at times you might want to refer to a file in an absolute location and this is supported by the project system.

When you add a file to a project, the project system detects whether the file is in the project directory. If a file is not in the project directory, the project system tries to make a relative path from the file to the project directory. If the file isn't relative to the project directory, the project system detects whether the file is relative to the **$(StudioDir)** directory; if so, the filename is defined using **$(StudioDir)**. If a file is not relative to the project directory or to **$(StudioDir)**, the full, absolute pathname is used.

The project system will allow (with a warning) duplicate files to be put into a project.

The project system uses a file's extension to determine the appropriate build action to perform on the file:

A file with the extension **.c** will be compiled by a C compiler.
A file with the extension **.cpp** or **.cxx** will be compiled by a C++ compiler.
A file with the extension **.s** or **.asm** will be compiled by an assembler.
A file with the object-file extension **.o** will be linked.
A file with the library-file extension **.a** will be linked.
A file with the extension **.xml** will be opened and its file type determined by the XML document type.
Files with other file extensions will not be compiled or linked.

You can modify this behavior by setting a file's **File Type** project option with the **Common** configuration selected, which enables files with non-standard extensions to be compiled by the project system.

Externally Built Executables

You can use an external build process for **Externally Built Executable** project types by setting the **Build Command** project option, for example to **make target**. Alternatively you can set command lines for specific build steps to compile/assemble and link. When you create an **Externally Built Executable** project type configurations will be created that create command lines for a variety of external tool chains.

Solution links

You can create links to existing project files from a solution, which enables you to create hierarchical builds. For example, you could have a solution that builds a library together with a stub test driver executable. You can link to that solution from your current solution by right-clicking the solution node of the **Project Explorer** and selecting **Add Existing Project**. Your current solution can then use the library built by the other project.

Session files

When you exit SEGGER Embedded Studio, details of your current session are stored in a *session file*. Session files are text files, with the file extension **emSession**, that contain details such as which files you have opened in the editor and what breakpoints you have set in the **Breakpoint** window.

Creating a project

You can create a new solution for each project or place multiple projects in an existing solution.

To create a new project in an existing solution:

1. Choose **Project** > **Add New Project**.
2. In the **New Project** wizard, select the type of project you wish to create and specify where it will be placed.
3. Ensure that **Add the project to current solution** is checked.
4. Click **OK** to go to next stage or **Cancel** to cancel the project's creation.

The project name must be unique to the solution and, ideally, the project directory should be relative to the solution directory. The project system will use the project directory as the *current directory* when it builds your project. Once complete, the **Project Explorer** displays the new solution, project, and files contained in the project. To add another project to the solution, repeat the above steps.

To create a new project in a new solution:

1. Choose **File** > **New Project** or press **Ctrl+Shift+N**.
2. Select the type of project you wish to create and where it will be placed.
3. Click **OK**.

Adding existing files to a project

You can add existing files to a project in a number of ways.

To add existing files to the active project:

Choose **Project** > **Add Existing File** or press **Ctrl+P, A**.

Using the **Open File** dialog, navigate to the directory containing the files and select the ones you wish to add to the project.

Click **OK**.

The selected files are added to the folders whose filter matches the extension of each of the files. If no filter matches a file's extension, the file is placed underneath the project node.

To add existing files to a specific project:

1. In the **Project Explorer**, right-click the project to which you wish to add a new file.
2. Choose **Add Existing File**.

To add existing files to a specific folder:

1. In the **Project Explorer**, right-click the folder to which you wish to add a new file.
2. Choose **Add Existing File**.

The files are added to the specified folder without using filter matching.

To create a dynamic folder:

1. In the **Project Explore**, right click on the project to which you wish to add a new folder.
2. Choose **New Folder...**.
3. Using the **New Folder** dialog name the folder and then show the dynamic folder options.
4. Specify the required **Source Folder** and the **Filter Specification**.

The files that match the filter specification in the source folder will appear in the newly created folder.

Adding new files to a project

You can add new files to a project in a number of ways.

To add new files to the active project:

Choose **Project** > **Add New File** or press **Ctrl+N**.

To add a new file to a project:

1. In the **Project Explorer**, right-click the project to which you wish to add a new file.
2. Choose **Add New File**.

When adding a new file, SEGGER Embedded Studio displays the **New File** dialog, from which you can choose the type of file to add, its filename, and where it will be stored. Once created, the new file is added to the folder whose filter matches the extension of the newly added file. If no filter matches the newly added file extension, the new file is placed underneath the project node.

To add new files to a folder:

1. In the **Project Explorer**, right-click the folder to which you wish to add a new file.
2. Choose **Add New File**.

The new file is added to the folder without using filter matching.

Removing a file, folder, project, or project link

You can remove whole projects, folders, or files from a project, or you can remove a project from a solution, using the **Remove** button on the **Project Explorer** toolbar. Note that removing a source file from a project does not remove it from disk.

To remove an item from the solution:

1. In the **Project Explorer**, select the item to remove.
2. Choose **Edit** > **Delete** or press **Del**.

or

1. In the **Project Explorer**, right-click the item to remove.
2. Choose **Remove**.

Building your application

SEGGER Embedded Studio builds your application using the resources and build rules it finds in your solution.

When SEGGER Embedded Studio builds your application, it tries to avoid building files that have not changed since they were last built. It does this by comparing the modification dates of the generated files with the modification dates of the dependent files together with the modification dates of the project options that pertain to the build. But if you are copying files, sometimes the modification dates may not be updated when the file is copiedin this instance, it is wise to use the **Rebuild** command rather than the **Build** command.

You can see the build rationale SEGGER Embedded Studio currently is using by setting the **Environment Options** > **Building** > **Show Build Information** environment option. To see the build commands themselves, set the **Environment Options** > **Building** > **Echo Build Command** environment option.

You may have a solution that contains several interdependent projects. Typically, you might have several executable projects and some library projects. The **Project Dependencies** dialog specifies the dependencies between projects and to see the effect of those dependencies on the solution build order. Note that dependencies can be set on a per-configuration basis, but the default is for dependencies to be defined in the **Common** configuration.

You will also notice that a new folder titled **Dependencies** has appeared in the **Project Explorer**. This folder contains the list of newly generated files and the files from which they were generated. To see if one of files can be decoded and displayed in the editor, right-click the file to see if the **View** command is available on the shortcut menu.

If you have the **Symbols** window open, it will be updated with the symbol and section information of all executable files built in the solution.

To generalize your builds, you can define macro values that are substituted when the project options are used. These macro values can be defined globally at the solution and project level, and can be defined on a per-configuration basis.

The combination of configurations, project options with inheritance, dependencies, and macros provides a very powerful build-management system. However, such systems can become complicated. To understand the implications of changing build settings, right-click a node in the **Project Explorer** and select **Options** to view a dialog that shows which macros and project options apply to that project node.

To build all projects in the solution:

1. Choose **Build** > **Build Solution** or press **Shift+F7**.

or

1. Right-click the solution in the **Project Explorer** window.
2. Choose **Build** from the shortcut menu.

To build a single project:

1. Select the required project in the **Project Explorer**.
2. Choose **Build** > **Build** or press **F7**.

or

1. Right-click the project in the **Project Explorer**.
2. Choose **Build**.

To compile a single file:

1. In the **Project Explorer**, click to select the source file to compile.
2. Choose **Build** > **Compile** or press **Ctrl+F7**.

or

1. In the **Project Explorer**, right-click the source file to compile.
2. Choose **Compile** from the shortcut menu.

Correcting errors after building

The results of a build are recorded in a **Build Log** that is displayed in the **Output** window. Errors are highlighted in red, warnings are highlighted in yellow. Double-clicking an error, warning, or note will move the insertion point to the line of source code that triggered that log entry.

You can move forward and backward through errors using **Search** > **Next Location** and **Search** > **Next Location**.

When you build a single project in a single configuration, the **Transcript** will display the memory used by the application and a summary for each memory area.

Creating variants using configurations

SEGGER Embedded Studio provides a facility to build projects in various configurations. Project configurations are used to create different software builds for your projects.

A configuration defines a set of project options. For example, the output of a compilation can be put into different directories, dependent upon the configuration. When you create a solution, some default project configurations are created.

Build configurations and their uses

Configurations are typically used to differentiate debug builds from release builds. For example, the compiler options for debug builds will differ from those of a release build: a debug build will set options so the project can be debugged easily, whereas a release build will enable optimization to reduce program size or to increase its speed. Configurations have other uses; for example, you can use configurations to produce variants of software, such as custom libraries for several different hardware variants.

Configurations inherit project options from other configurations. This provides a single point of change for definitions common to several configurations. A particular project option can be overridden in a particular configuration to provide configuration-specific settings.

When a solution is created, two configurations are generated **Debug** and **Release** and you can create additional configurations by choosing **Build** > **Build Configurations**. Before you build, ensure that the appropriate configuration is set using **Build** > **Set Active Build Configuration** or, alternatively, the **Active Configuration** combo box in the **Project Explorer**.

Selecting a configuration

To set the configuration that affects your building and debugging, use the combo box in the **Project Explorer** or select **Build** > **Set Active Build Configuration**

Creating a configuration

To create your own configurations, select **Build** > **Build Configurations** to invoke the **Configurations** dialog. The **New** button will produce a dialog allowing you to name your configuration. You can now specify the existing configurations from which your new configuration will inherit values.

Deleting a configuration

You can delete a configuration by selecting it and clicking the **Remove** button. This deletion cannot be undone or canceled, so beware.

Private configurations

Some configurations are defined purely for inheriting and, as such, should not appear in the **Build** combo box. When you select a configuration in the **Configuration** dialog, you can choose to hide that configuration.

Project options

For solutions, projects, folders, and files, project options can be defined that are used by the project system in the build process. These project options can be viewed and modified by using the **Options** dialog in conjunction with the **Project Explorer**.

Some project options are only applicable to a given item type. For example, linker project options are only applicable to a project that builds an executable file. However, other project options can be applied either at the file, project, or solution project node. For example, a compiler project option can be applied to a solution, project, or individual file. By setting a project option at the solution level, you enable all files of the solution to use that project option's value.

Unique project options

A unique project option has *one* value. When a build is done, the value of a unique project option is the first one defined in the project hierarchy. For example, the **Treat Warnings As Errors** project option could be set to **Yes** at the solution level, which would then be applicable to every file in the solution that is compiled, assembled, and linked. You can then selectively define project options for other project items. For example, a particular source file may have warnings you decide are allowable, so you set the **Treat Warnings As Errors** to **No** for that particular file.

```
solution  Treat Warnings As Errors = Yes
    project1  Treat Warnings As Errors = Yes
        file1  Treat Warnings As Errors = Yes
        file2  Treat Warnings As Errors = No
    project2  Treat Warnings As Errors = No
        file1  Treat Warnings As Errors = No
        file2  Treat Warnings As Errors = Yes
```

In the above example, the files will be compiled with these values for **Treat Warnings As Errors**:

project1/file1	Yes
project1/file2	No
project2/file1	No
project2/file2	Yes

Aggregate project options

An aggregating project option collects all the values defined for it in the project hierarchy. For example, when a C file is compiled, the **Preprocessor Definitions** project option will take all the values defined at the file, project, and solution levels.

```
solution  Preprocessor Definitions = SolutionDef
```

```
    project1  Preprocessor Definitions =
        file1  Preprocessor Definitions =
        file2  Preprocessor Definitions = File1Def
    project2  Preprocessor Definitions = ProjectDef
        file1  Preprocessor Definitions =
        file2  Preprocessor Definitions = File2Def
```

In the above example, the files will be compiled with these preprocessor definitions:

project1/file1	SolutionDef
project1/file2	SolutionDef, File1Def
project2/file1	SolutionDef, ProjectDef
project2/file2	SolutionDef, ProjectDef, File2Def

Configurations and project options

Project options are defined for a configuration so you can have different values for a project option for different builds. A given configuration can inherit the project options of other configurations. When the project system requires a project option value, it checks for the existence of the project option value in the current configuration and then in the set of inherited configurations. You can specify the set of inherited configurations using the **Configurations** dialog.

A special configuration named **Common** is always inherited by a configuration. The **Common** configuration allows you to set project options that will apply to all configurations you create. If you are modifying a project option of your project, you almost certainly want each configuration to inherit it, so ensure that the **Common** configuration is selected.

If the project option is unique, the build system will use the one defined for the particular configuration. If the project option isn't defined for this configuration, the build system uses an arbitrary one from the set of inherited configurations.

If the option is still undefined, the build system uses the value for the **Common** configuration. If it is still undefined, the build system tries to find the value in the next higher level of the project hierarchy.

```
solution [Common]  Preprocessor Definitions = CommonSolutionDef

solution [Debug]  Preprocessor Definitions = DebugSolutionDef

solution [Release]  Preprocessor Definitions = ReleaseSolutionDef

    project1 - Preprocessor Definitions =

        file1 - Preprocessor Definitions =

        file2 [Common]  Preprocessor Definitions = CommonFile1Def

        file2 [Debug]  Preprocessor Definitions = DebugFile1Def

    project2 [Common]  Preprocessor Definitions = ProjectDef

        file1  Preprocessor Definitions =

        file2 [Common] - Preprocessor Definitions = File2Def
```

In the above example, the files will be compiled with these preprocessor definitions when in **Debug** configuration

File	Setting
project1/file1	CommonSolutionDef, DebugSolutionDef
project1/file2	CommonSolutionDef, DebugSolutionDef,CommonFile1Def, DebugFile1Def
project2/file1	CommonSolutionDef, DebugSolutionDef, ProjectDef

project2/file2	ComonSolutionDef, DebugSolutionDef, ProjectDef, File2Def

and the files will be compiled with these **Preprocessor Definitions** when in **Release** configuration:

File	Setting
project1/file1	CommonSolutionDef, ReleaseSolutionDef
project1/file2	CommonSolutionDef, ReleaseSolutionDef, CommonFile1Def
project2/file1	CommonSolutionDef, ReleaseSolutionDef, ProjectDef
project2/file2	ComonSolutionDef, ReleaseSolutionDef, ProjectDef, File2Def

Project macros

You can use macros to modify the way the project system refers to files.

Macros are divided into four classes:

System macros defined by SEGGER Embedded Studio relay information about the environment, such as paths to common directories.
Global macros are saved in the environment and are shared across all solutions and projects. Typically, you would set up paths to libraries and any external items here.
Project macros are saved as project options in the project file and can define values specific to the solution or project in which they are defined.
Build macros are generated by the project system when you build your project.

System macros

System macros are defined by SEGGER Embedded Studio itself and as such are read-only. System macros can be used in project options, environment settings and to refer to files. See **System macros list** for the list of System macros.

Global macros

Global macros are store in the environment option **Build Macros**.

To define a global macro:

1. Use **Tools** > **Options** to show the environment options dialog.
2. In the **Environment Options** dialog's **Building** group, select the **Build Macros** option.
3. Click the ellipsis button on the right.
4. Set the macro using the syntax *name* = *replacement text*.

Project macros

To define a project macro:

To set the project macros:

1. Select the appropriate solution/project in the **Project Explorer**.
2. Use **Project** > **Options** to show the project options dialog.
3. In the **Project Options** dialog's **General Options** group, select the **Macros** option.
4. Click the ellipsis button on the right.
5. Set the macro using the syntax *name* = *replacement text*.

Build macros

Build macros are defined by the project system for a build of a given project node. See **Build macros list** for the list of build macros.

Using macros

You can use a macro for a project option or environment setting by using the $(macro) syntax. For example, the **Object File Name** option has a default value of `$(IntDir)/$(InputName)$(OBJ)`.

You can also specify a default value for a macro if it is undefined using the $(macro:default) syntax. For example, `$(MyMacro:0)` would expand to `0` if the macro `MyMacro` has not been defined.

Dependencies and build order

You can set up dependency relationships between projects using the **Project Dependencies** dialog. Project dependencies make it possible to build solutions in the correct order and, where the target permits, to load and delete applications and libraries in the correct order. A typical usage of project dependencies is to make an executable project dependent upon a library executable. When you elect to build the executable, the build system will ensure that the library it depends upon is up to date. In the case of a dependent library, the output file of the library build is supplied as an input to the executable build, so you don't have to worry about it.

Project dependencies are stored as project options and, as such, can be defined differently based upon the selected configuration. You almost always want project dependencies to be independent of the configuration, so the **Project Dependencies** dialog selects the **Common** configuration by default.

To make one project dependent upon another:

1. Choose **Project** > **Project Dependencies**.
2. From the **Project** dropdown, select the target project that depends upon other projects.
3. In the **Depends Upon** list box, select the projects the target project depends upon and deselect the projects it does not depend upon.

Some items in the **Depends Upon** list box may be dimmed, indicating that a circular dependency would result if any of those projects were selected. In this way, SEGGER Embedded Studio prevents you from constructing circular dependencies using the **Project Dependencies** dialog.

If your target supports loading multiple projects, the **Build Order** also reflects the order in which projects are loaded onto the target. Projects will load, in order, from top to bottom. Generally, libraries need to be loaded before the applications that use them, and you can ensure this happens by making the application dependent upon the library. With this dependency set, the library gets built and loaded before the application does.

Applications are deleted from a target in reverse of their build order; in this way, applications are removed before the libraries on which they depend.

Linking and section placement

Executable programs consist of a number of sections. Typically, there are program sections for code, initialized data, and zeroed data. There is often more than one code section and they must be placed at specific addresses in memory.

To describe how the program sections of your program are positioned in memory, the SEGGER Embedded Studio project system uses *memory-map* files and *section-placement* files. These XML-formatted files are described in **Memory Map file format** and **Section Placement file format**. They can be edited with the SEGGER Embedded Studio text editor. The memory-map file specifies the start address and size of target memory segments. The section-placement file specifies where to place program sections in the target's memory segments. Separating the memory map from the section-placement scheme enables a single hardware description to be shared across projects and also enables a project to be built for a variety of hardware descriptions.

For example, a memory-map file representing a device with two memory segments called **FLASH** and **SRAM** could look something like this in the memory-map editor.

```
<Root name="Device1">
  <MemorySegment name="FLASH" start="0x10000000" size="0x10000" />
  <MemorySegment name="SRAM" start="0x20000000" size="0x1000" />
```

A corresponding section-placement file will refer to the memory segments of the memory-map file and will list the sections to be placed in those segments. This is done by using a memory-segment name in the section-placement file that matches the corresponding memory-segment name in the memory-map file.

For example, a section-placement file that places a section called **.stack** in the **SRAM** segment and the **.vectors** and **.text** sections in the **FLASH** segment would look like this:

```
<Root name="Flash Section Placement">
  <MemorySegment name="FLASH" >
    <ProgramSection name=".vectors" load="Yes" />
    <ProgramSection name=".text" load="Yes" />
  </MemorySegment>
  <MemorySegment name="SRAM" >
   <ProgramSection name=".stack" load="No" />
  </MemorySegment>
</Root>
```

Note that the order of section placement within a segment is top down; in this example **.vectors** is placed at lower addresses than **.text**.

The memory-map file and section-placement file to use for linkage can be included as a part of the project or, alternatively, they can be specified in the project's **linker options**.

You can create a new program section using either the assembler or the compiler. For the C/C++ compiler, this can be achieved using **__attribute__** on declarations. For example:

```
void foobar(void) __attribute__ ((section(".foo")));
```

This will allocate **foobar** in the section called **.foo**. Alternatively, you can specify the names for the code, constant, data, and zeroed-data sections of an entire compilation unit by using the **Section Options** options.

You can now place the section into the section placement file using the editor so that it will be located after the vectors sections as follows:

```
<Root name="Flash Section Placement">
  <MemorySegment name="FLASH">
    <ProgramSection name=".vectors" load="Yes" />
    <ProgramSection name=".foo" load="Yes" />
    <ProgramSection name=".text" load="Yes" />
  </MemorySegment>
  <MemorySegment name="SRAM">
   <ProgramSection name=".stack" load="No" />
  </MemorySegment>
</Root>
```

If you are modifying a section-placement file that is supplied in the SEGGER Embedded Studio distribution, you will need to import it into your project using the **Project Explorer**.

Sections containing code and constant data should have their **load** project option set to **Yes**. Some sections don't require any loading, such as stack sections and zeroed-data sections; such sections should have their **load** project option set to **No**.

Some sections that are loaded then need to be copied to sections that aren't yet loaded. This is required for initialized data sections and to copy code from slow memory regions to faster ones. To do this, the **runin** attribute should contain the name of a section in the section-placement file to which the section will be copied.

For example, initialized data is loaded into the **.data_load** section and then is copied into the **.data_run** section using:

```
<Root name="Flash Section Placement">
  <MemorySegment name="FLASH">
    <ProgramSection name=".vectors" load="Yes" />
    <ProgramSection name=".text" load="Yes" />
    <ProgramSection name=".data_load" load="Yes" runin="data_run" />
  </MemorySegment>
  <MemorySegment name="SRAM">
    <ProgramSection name=".data_run" load="No" />
    <ProgramSection name=".stack" load="No" />
  </MemorySegment>
</Root>
```

The startup code will need to copy the contents of the **.data_load** section to the **.data_run** section. To enable this, symbols are generated marking the start and end addresses of each section. For each section, a start symbol called __*section-name*_**start**__ and an end symbol called __*section-name*_**end**__ are generated. These symbols can be used to copy the sections from their load positions to their run positions.

For example, the **.data_load** section can be copied to the **data_run** section using the following call to `memcpy`.

```
/* Section image located in flash */
extern const unsigned char __data_load_start__[];
extern const unsigned char __data_load_end__[];
```

```
/* Where to locate the section image in RAM. */
extern unsigned char __data_run_start__[];
extern unsigned char __data_run_end__[];

/* Copy image from flash to RAM. */
memcpy(__data_run_start__,
       __data_load_start__,
       __data_load_end__ - __data_load_start__);
```

Using source control

Source control is an essential tool for individuals or development teams. SEGGER Embedded Studio integrates with several popular source-control systems to provide this feature for files in your SEGGER Embedded Studio projects.

Source-control capability is implemented by a number of third-party providers, but the set of functions provided by SEGGER Embedded Studio aims to be provider independent.

Source control capabilities

The source-control integration capability provides:

- Connecting to the source-control *repository* and mapping files in the SEGGER Embedded Studio project to those in source control.
- Showing the source-control status of files in the project.
- Adding files in the project to source control.
- Fetching files in the project from source control.
- Optionally locking and unlocking files in the project for editing.
- Comparing a file in the project with the latest version in source control.
- Updating a file in the project by merging changes from the latest version in source control.
- Committing changes made to project files into source control.

Configuring source-control providers

SEGGER Embedded Studio supports Subversion, Git, and Mercurial as source-control systems. To enable SEGGER Embedded Studio to utilize source-control features, you need to install, on your operating system, the appropriate command line client for the source-control systems that you will use.

Once you have installed the command line client, you must configure SEGGER Embedded Studio to use it.

To configure Subversion:

1. Choose **Tools** > **Options** or press **Alt+,**.
2. Select the **Source Control** category in the options dialog.
3. Set the **Executable** environment option of the **Subversion Options** group to point to Subversion `svn` command. On Windows operating systems, the Subversion command is `svn.exe`.

To configure Git:

1. Choose **Tools** > **Options** or press **Alt+,**.
2. Select the **Source Control** category in the options dialog.
3. Set the **Executable** environment option of the **Git Options** group to point to Git `git` command. On Windows operating systems, the Git command is `git.exe`.

To configure Mercurial:

1. Choose **Tools** > **Options** or press **Alt+,**.
2. Select the **Source Control** category in the options dialog.
3. Set the **Executable** environment option of the **Mercurial Options** group to point to Git `hg` command. On Windows operating systems, the Git command is `hg.exe`.

Connecting to the source-control system

When SEGGER Embedded Studio loads a project, it examines the file system folder that contains the project to determine the source-control system the project uses. If SEGGER Embedded Studio cannot determine, from the file system, the source-control system in use, it disables source-control integration.

That is, if you have not set up the paths to the source-control command line clients, even if a working copy exists and the appropriate command line client is installed, SEGGER Embedded Studio cannot establish source-control integration for the project.

User credentials

You can set the credentials that the source-control system uses, for commands that require credentials, using **VCS** > **Options** > **Configure**. From here you can set the user name and password. These details are saved to the session file (the password is encrypted) so you won't need to specify this information each time the project is loaded.

Note

SEGGER Embedded Studio has no facility to create repositories from scratch, nor to clone, pull, or checkout repositories to a working copy: it is your responsibility to create a working copy outside of SEGGER Embedded Studio using your selected command-line client or Windows Explorer extension.

The "Tortoise" products are a popular set of tools to provide source-control facilities in the Windows shell. Use Google to find **TortoiseSVN**, **TortoiseGit**, and **TortoiseHG** and see if you like them.

File source-control status

Determining the source-control status of a file can be expensive for large repositories, so SEGGER Embedded Studio updates the source-control status in the background. Priority is given to items that are displayed.

A file will be in one of the following states:

Clean:The file is in source control and matches the tip revision.
Not Controlled:The file is not in source control.
Conflicted:The file is in conflict with changes made to the repository.
Locked:The file is locked.
Update Available:The file is older than the most-recent version in source control.
Added:The file is scheduled to be added to the repository.
Removed:The file is scheduled to be removed from the repository.

If the file has been modified, its status is displayed in red in the **Project Explorer**. Note that if a file is not under the local root, it will not have a source-control status.

You can reset any stored source-control file status by choosing **VCS** > **Refresh**.

Source-control operations

Source-control operations can be performed on single files or recursively on multiple files in the **Project Explorer** hierarchy. Single-file operations are available on the **Source Control** toolbar and on the text editor's shortcut menu. All operations are available using the **VCS** menu. The operations are described in terms of the **Project Explorer** shortcut menu.

Adding files to source control

To add files to the source-control system:

1. In the **Project Explorer**, select the file to add. If you select a folder, project, or solution, any eligible child items will also be added to source control.
2. choose **Source Control** > **Add** or press **Ctrl+R, A**.
3. The dialog will list the files that can be added.
4. In that dialog, you can deselect any files you don't want to add to source control.
5. Click **Add**.

Note

Files are scheduled to be added to source control and will only be committed to source control (and seen by others) when you commit the file.

Enabling the **VCS** > **Options** > **Add Immediately** option will bypass the dialog and immediately add (but not commit) the files.

Updating files

To update files from source control:

1. In the **Project Explorer**, select the file to update. If you select a folder, project, or solution, any eligible child items will also be updated from source control.
2. choose **Source Control** > **Update** or press **Ctrl+R, U**.
3. The dialog will list the files that can be updated.
4. In that dialog, you can deselect any files you don't want to update from source control.
5. Click **Update**.

Note

Enabling the **VCS** > **Options** > **Update Immediately** option will bypass the dialog and immediately update the files.

Committing files

To commit files:

1. In the **Project Explorer**, select the file to commit. If you select a folder, project, or solution, any eligible child items will also be committed.
2. Choose **Source Control** > **Commit** or press **Ctrl+R, C**.
3. The dialog will list the files that can be committed.
4. In that dialog, you can deselect any files you don't want to commit and enter an optional comment.
5. Click **Commit**.

Note

Enabling the **VCS** > **Options** > **Commit Immediately** option will bypass the dialog and immediately commit the files without a comment.

Reverting files

To revert files:

1. In the **Project Explorer**, select the file to revert. If you select a folder, project, or solution, any eligible child items will also be reverted.
2. Choose **Source Control** > **Revert** or press **Ctrl+R, V**.
3. The dialog will list the files that can be reverted.
4. In that dialog, you can deselect any files you don't want to revert.
5. Click **Revert**.

Note

Enabling the **VCS** > **Options** > **Revert Immediately** option will bypass the dialog and immediately revert files.

Locking files

To lock files:

1. In the **Project Explorer**, select the file to lock. If you select a folder, project, or solution, any eligible child items will also be locked.
2. Choose **Source Control** > **Lock** or press **Ctrl+R, L**.
3. The dialog will list the files that can be locked.
4. In that dialog, you can deselect any files you don't want to lock and enter an optional comment.
5. Click **Lock**.

Note

Enabling the **VCS** > **Options** > **Lock Immediately** option will bypass the dialog and immediately lock files without a comment.

Unlocking files

To unlock files:

1. In the **Project Explorer**, select the file to lock. If you select a folder, project, or solution, any eligible child items will also be unlocked.
2. Choose **Source Control** > **Unlock** or press **Ctrl+R, N**.
3. The dialog will list the files that can be unlocked.
4. In that dialog, you can deselect any files you don't want to unlock.
5. Click **Unlock**.

Note

Enabling the **VCS** > **Options** > **Unlock Immediately** option will bypass the dialog and immediately unlock files.

Removing files from source control

To remove files from source control:

1. In the **Project Explorer**, select the file to remove. If you select a folder, project, or solution, any eligible child items will also be removed.
2. choose **Source Control** > **Remove** or press **Ctrl+R, R**.
3. The dialog will list the files that can be removed.
4. In that dialog, you can deselect any files you don't want to remove.
5. Click **Remove**.

Note

Files are scheduled to be removed from source control and will still be and seen by others, giving you the opportunity to revert the removal. When you commit the file, the file is removed from source control.

Enabling the **VCS** > **Options** > **Remove Immediately** option will bypass the dialog and immediately remove (but not commit) files.

Showing differences between files

To show the differences between the file in the project and the version checked into source control, do the following:

1. In the **Project Explorer**, right-click the file.
2. From the shortcut menu, choose **Source Control** > **Show Differences**.

You can use an external diff tool in preference to the built-in SEGGER Embedded Studio diff tool. To define the diff command line SEGGER Embedded Studio generates, choose **Tools** > **Options** > **Source Control** > **Diff Command Line**. The command line is defined as a list of strings to avoid problems with spaces in arguments. The diff command line can contain the following macros:

$(localfile):The filename of the file in the project.
$(remotefile):The filename of the latest version of the file in source control.
$(localname):A display name for $(localfile).
$(remotename):A display name for $(remotefile).

Source-control properties

When a file in the project is in source control, the **Properties** window shows the following properties in the **Source Control Options** group:

Property	Description
SEGGER Embedded Studio Status	The source-control status of working copy as viewed by SEGGER Embedded Studio.
last Author	The author of the file's head revision.
Path: Relative	The item's path relative to the repository root.
Path: Repository	The pathname of the file in the source-control system, typically a URL.
Path: Working Copy	The pathname of the file in the working copy.
Provider	The name of the source-control system managing this file.
Provider Status	The status of the file as reported by the source-control provider.
Revision: Local	The revision number/name of the local file.
Revision: Remote	The revision number/name of the most-recent version in source control.
Status: In Conflict?	If **Yes**, updates merged into the file using **Update** conflict with the changes you made locally; if **No**, the file is not locked. When conflicted, must resolve the conflicts and mark them `Resolved` before committing the file.
Status: Locked?	If **Yes**, the file is lock by you; if **No**, the file is not locked.
Status: Modified?	If **Yes**, the checked-out file differs from the version in the source control system; if **No**, they are identical.
Status: Update Available?	If **Yes**, the file in the project location is an old version compared to the latest version in the source-control systemuse **Update** to merge in the latest changes.

Subversion provider

The Subversion source-control provider has been tested with SVN 1.4.3.

Provider-specific options

The following environment options are supported:

Property	Description
Executable	The path to the `svn` executable.
Lock Supported	If **Yes**, check out and undo check out operations are supported. Check out will issue the `svn lock` command; check in and undo check out will issue the `svn unlock` command.
Authentication	Selects whether authentication (user name and password) is sent with every command.
Show Updates	Selects whether the update (`-u` flag) is sent with status requests in order to show that new versions are available in the repository. Note that this requires a live connection to the repository: if you are working without a network connection to your repository, you can disable this switch and continue to enjoy source control status information in the **Project Explorer** and **Pending Changes** windows.

Connecting to the source-control system

When connecting to source control, the provider checks if the local root is in SVN control. If this is the case, the local and remote root will be set accordingly. If the local root is not in SVN control after you have set the remote root, a `svn checkout -N` command will be issued to make the local root SVN controlled. This command will also copy any files in the remote root to the local root.

The user name and password you enter will be supplied with each svn command the provider issues.

Source control operations

The SEGGER Embedded Studio source-control operations are implemented using Subversion commands. Mapping SEGGER Embedded Studio source-control operations to Subversion source-control operations is straightforward:

Operation	Command
Commit	`svn commit` for the file, with optional comment.
Update	`svn update` for each file.

Revert	`svn revert` for each file.
Resolved	`svn resolved` for each file.
Lock	`svn lock` for each file, with optional comment.
Unlock	`svn unlock` for each file.
Add	`svn add` for each file.
Remove	`svn remove` for each file.
Source Control Explorer	`svn list` with a remote root. `svn mkdir` to create directories in the repository.

CVS provider

The CVS source-control provider has been tested with CVSNT 2.5.03. The CVS source-control provider uses the CVS `rls` command to browse the repositorythis command is implemented in CVS 1.12 but usage of `.` as the root of the module name is not supported.

Provider-specific options

The following environment options are supported:

Property	Description
CVSROOT	The CVSROOT value to access the repository.
Edit/Unedit Supported	If **Yes**, **Check Out** and **Undo Check Out** commands are supported. Any check-out operation will issue the `cvs edit` command; any check-in or undo-check-out operation will issue the `cvs unedit` command; the status operation will issue the `cvs ss` command.
Executable	The path to the `cvs` executable.
Login/Logout Required	If **Yes**, **Connect** will issue the `cvs login` command.

Connecting to the source-control system

When connecting to source control, the provider checks if the local root is in CVS control. If this is the case, the local and remote root will be set accordingly. If the local root is not in CVS control after you have set the remote root, a `cvs checkout -l -d` command will be issued to make the local root CVS controlled. This command will also copy any files in the remote root to the local root.

Source-control operations

The SEGGER Embedded Studio source-control operations have been implemented using CVS commands. There are no multiple-file operations, each operation is done on a single file and committed as part of the operation.

Operation	Command
Get Status	`cvs status` and optional `cvs editors` for local directories in CVS control. `cvs rls -e` for directories in the repository.
Add To Source Control	`cvs add` for each directory not in CVS control. `cvs add` for the file. `cvs commit` for the file and directories.
Get Latest	`cvs update -l -d` for each directory not in CVS control. `cvs update` to merge the local file. `cvs update -C` to overwrite the local file.

Check Out	Optional `cvs update -C` to get the latest version. `cvs edit` to lock the file.
Undo Check Out	`cvs unedit` to unlock the file. Optional `cvs update` to get the latest version.
Check In	`cvs commit` for the file.
Source Control Explorer	`cvs rls -e` with a remote root starting with .. `cvs import` to create directories in the repository.

Package management

Additional target-support functions can be added to, and removed from, SEGGER Embedded Studio with *packages*.

A SEGGER Embedded Studio package is an archive file containing a collection of target-support files. Installing a package involves copying the files it contains to an appropriate destination directory and registering the package with SEGGER Embedded Studio's package system. Keeping target-support files separate from the main SEGGER Embedded Studio installation allows us to support new hardware and issue bug fixes for existing hardware-support files between SEGGER Embedded Studio releases, and it allows third parties to develop their own support packages.

Installing packages

Use the **Package Manager** to automate the download, installation, upgrade and removal of packages.

To activate the Package Manager:

Choose **Tools** > **Manage Packages**.

In some situations, such as using SEGGER Embedded Studio on a computer without Internet access or when you want to install packages that are not on the website, you cannot use the **Package Manager** to install packages and it will be necessary to manually install them.

To manually install a package:

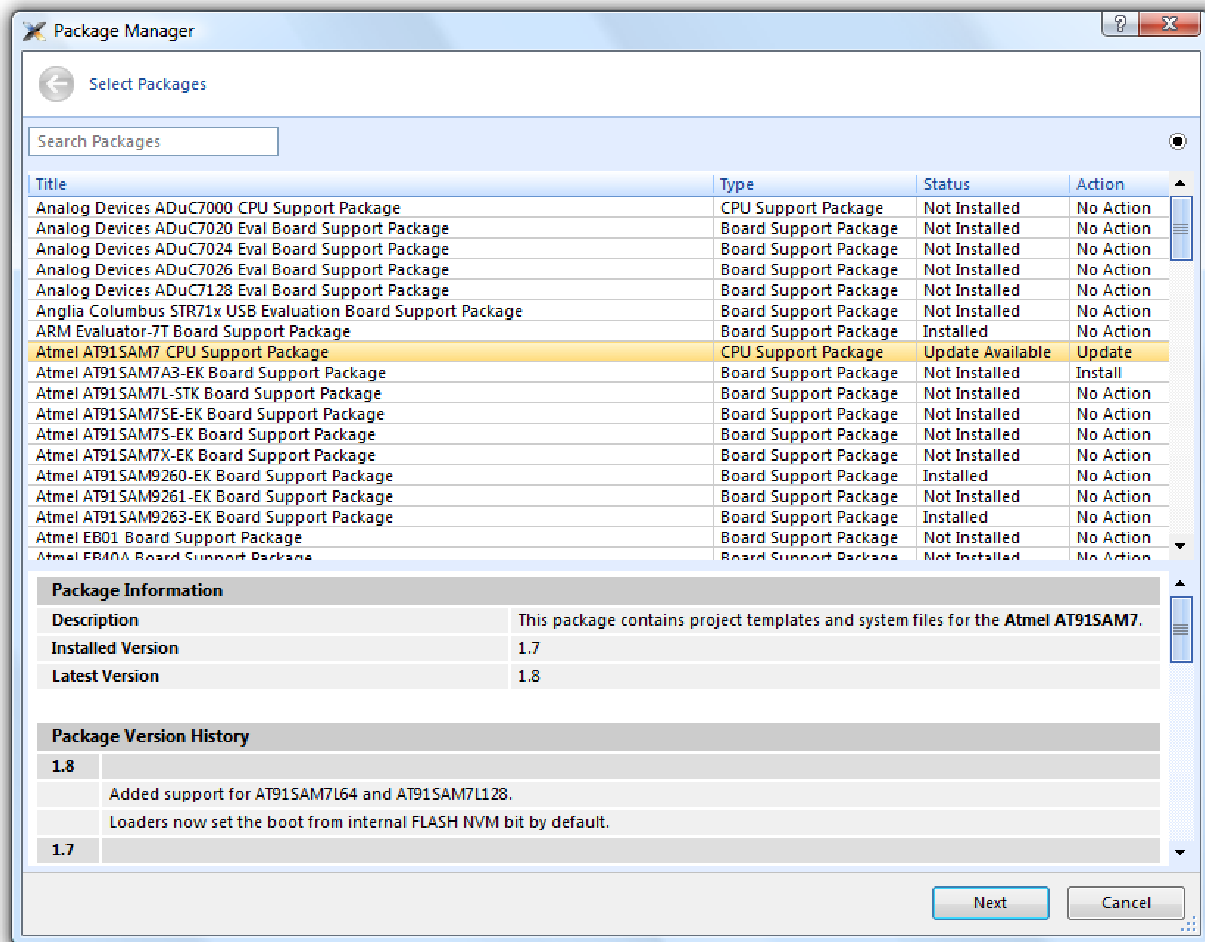
1. Choose **Tools** > **Manually Install Packages**.
2. Select one or more package files you want to install.
3. Click **Open** to install the packages.

Choose **Tools** > **Show Installed Packages** to see more information on the installed packages.

The **Package Manager** window will remove manually installed packages.

The package manager

The **Package Manager** manages the support packages installed on your system. It lists the available packages, shows the installed packages, and allows you to install, update, reinstall, and remove them.



To activate the Package Manager:

Choose **Tools** > **Manage Packages**.

Filtering the package list

By default, the **Package Manager** lists all available and installed packages. You can filter the displayed packages in a number of ways.

To filter by package status:

Click on the disclosure icon near the top-right corner of the dialog.

Use the pop-up menu to choose how to filter the list of packages.

The list-filter choices are:

Display All Show all packages irrespective of their status.

Display Not Installed Show packages that are available but are not currently installed.

Display Installed Only show packages that are installed.

Display Updates Only show packages that are installed but are not up-to-date because a newer version is available.

You can also filter the list of packages by the text in the package's title and documentation.

To filter packages by keyword:

Type the keyword into the **Search Packages** box at the top-left corner of the dialog.

Installing a package

The package-installation operation downloads a package to **$(PackagesDir)/downloads**, if it has not been downloaded already, and unpacks the files contained within the package to their destination directory.

To install a package:

1. Choose **Tools** > **Package Manager** and set the status filter to **Display Not Installed**.
2. Select the package or packages you wish to install.
3. Right-click the selected packages and choose **Install Selected Packages** from the shortcut menu.
4. Click **Next**; you will be see the actions the **Package Manager** is about to carry out.
5. Click **Next** and the **Package Manager** will install the selected packages.
6. When installation is complete, click **Finish** to close the **Package Manager**.

Updating a package

The package-update operation first removes existing package files, then it downloads the updated package to **$(PackagesDir)/downloads** and unpacks the files contained within the package to their destination directory.

To update a package:

1. Choose **Tools** > **Package Manager** and set the status filter to **Display Updates**.
2. Select the package or packages you wish to update.
3. Right-click the selected packages and choose **Update Selected Packages** from the shortcut menu.
4. Click **Next**; you will see the actions the **Package Manager** is about to carry out.
5. Click **Next** and the **Package Manager** will update the package(s).
6. When the update is complete, click **Finish** to close the **Package Manager**.

Removing a package

The package-remove operation removes all the files that were extracted when the package was installed.

To remove a package:

1. Choose **Tools** > **Package Manager** and set the status filter to **Display Installed**.
2. Select the package or packages you wish to remove.
3. Right-click the selected packages and choose **Remove Selected Packages** from the shortcut menu.
4. Click **Next**; you will see the actions the **Package Manager** is about to carry out.
5. Click **Next** and the **Package Manager** will remove the package(s).
6. When the operation is complete, click **Finish** to close the **Package Manager**.

Reinstalling a package

The package-reinstall operation carries out a package-remove operation followed by a package-install operation.

To reinstall a package:

1. Choose **Tools** > **Package Manager** and set the status filter to **Display Installed**.
2. Select the package or packages you wish to reinstall.
3. Right-click the packages to reinstall and choose **Reinstall Selected Packages** from the shortcut menu.
4. Click **Next**; you will see the actions the **Package Manager** is about to carry out.
5. Click **Next** and the **Package Manager** will reinstall the packages.
6. When the operation is complete, click **Finish** to close the **Package Manager**.

Exploring your application

In this section, we discuss the SEGGER Embedded Studio tools that help you examine how your application is built.

Project explorer

The **Project Explorer** is the user interface of the SEGGER Embedded Studio project system. It organizes your projects and files and provides access to the commands that operate on them. A toolbar at the top of the window offers quick access to commonly used commands for the selected project node or the active project. Right-click to reveal a shortcut menu with a larger set of commands that will work on the selected project node, ignoring the active project.

The selected project node determines what operations you can perform. For example, the **Compile** operation will compile a single file if a file project node is selected; if a folder project node is selected, each of the files in the folder are compiled.

You can select project nodes by clicking them in the **Project Explorer**. Additionally, as you switch between files in the editor, the selection in the **Project Explorer** changes to highlight the file you're editing.

To activate the Project Explorer:

Choose **View** > **Project Explorer** or press **Ctrl+Alt+P**.

Left-click operations

The following operations are available in the **Project Explorer** with a left-click of the mouse:

Action	Description
Single click	Select the node. If the node is already selected and is a solution, project, or folder node, a rename editor appears.
Double click	Double-clicking a solution node or folder node will reveal or hide the node's children. Double-clicking a project node selects it as the active project. Double-clicking a file opens the file with the default editor for that file's type.

Toolbar commands

The following buttons are on the toolbar:

Button	Description
	Add a new file to the active project using the **New File** dialog.
	Add existing files to the active project.

	Remove files, folders, projects, and links from the project.
	Create a new folder in the active project.
	Menu of build operations.
	Disassemble the active project.
	Menu of **Project Explorer** options.
	Display the properties dialog for the selected item.

Shortcut menu commands

The shortcut menu, displayed by right-clicking, contains the commands listed below.

For solutions:

Item	Description
Build and Batch Build	Build all projects under the solution in the current or batch build configuration.
Rebuild and Batch Rebuild	Rebuild all projects under the solution in the current or batch build configuration.
Clean and Batch Clean	Remove all output and intermediate build files for the projects under the solution in the current or batch build configuration.
Export Build and Batch Export Build	Create an editor with the build commands for the projects under the solution in the current or batch build configuration.
Add New Project	Add a new project to the solution.
Add Existing Project	Create a link from an existing solution to this solution.
Paste	Paste a copied project into the solution.
Remove	Remove the link to another solution from the solution.
Rename	Rename the solution node.
Source Control Operations	Source-control operations on the project file and recursive operations on all files in the solution.
Edit Solution As Text	Create an editor containing the project file.
Save Solution As	Change the filename of the project filenote that the saved project file is not reloaded.
Properties	Show the **Properties** dialog with the solution node selected.

For projects:

Item	Description
Build and Batch Build	Build the project in the current or batch build configuration.
Rebuild and Batch Rebuild	Reuild the project in the current or batch build configuration.
Clean and Batch Clean	Remove all output and intermediate build files for the project in the current or batch build configuration.
Export Build and Batch Export Build	Create an editor with the build commands for the project in the current or batch build configuration.
Link	Perform the project node build operation: link for an Executable project type, archive for a Library project type, and the combine command for a Combining project type.
Set As Active Project	Set the project to be the active project.
Debugging Commands	For Executable and Externally Built Executable project types, the following debugging operations are available on the project node: **Start Debugging**, **Step Into Debugging**, **Reset And Debug**, **Start Without Debugging**, **Attach Debugger**, and **Verify**.
Memory-Map Commands	For Executable project types that don't have memory-map files in the project and have the memory-map file project option set, there are commands to view the memory-map file and to import it into the project.
Section-Placement Commands	For Executable project types that don't have section-placement files in the project but have the section-placement file project option set, there are commands to view the section-placement file and to import it into the project.
Target Processor	For Executable and Externally Built Executable project types that have a Target Processor option group, the selected target can be changed.
Add New File	Add a new file to the project.
Add Existing File	Add an existing file to the project.
New Folder	Create a new folder in the project.
Cut	Cut the project from the solution.
Copy	Copy the project from the solution.
Paste	Paste a copied folder or file into the project.
Remove	Remove the project from the solution.
Rename	Rename the project.

Source Control Operations	Source-control, recursive operations on all files in the project.
Find in Project Files	Run **Find in Files** in the project directory.
Properties	Show the **Project Manager** dialog and select the project node.

For folders:

Item	Description
Add New File	Add a new file to the folder.
Add Existing File	Add an existing file to the folder.
New Folder	Create a new folder in the folder.
Cut	Cut the folder from the project or folder.
Copy	Copy the folder from the project or folder.
Paste	Paste a copied folder or file into the folder.
Remove	Remove the folder from the project or folder.
Rename	Rename the folder.
Source Control Operations	Source-control recursive operations on all files in the folder.
Compile	Compile each file in the folder.
Properties	Show the properties dialog with the folder node selected.

For files:

Item	Description
Open	Edit the file with the default editor for the file's type.
Open With	Edit the file with a selected editor. You can choose from the **Binary Editor**, **Text Editor**, and **Web Browser**.
Select in File Explorer	Create a operating system file system window with the file selected.
Compile	Compile the file.
Export Build	Create an editor window containing the commands to compile the file in the active build configuration.
Exclude From Build	Set the **Exclude From Build** option to **Yes** for this project node in the active build configuration.
Disassemble	Disassemble the output file of the compile into an editor window.
Preprocess	Run the C preprocessor on the file and show the output in an editor window.
Cut	Cut the file from the project or folder.

Copy	Copy the file from the project or folder.
Remove	Remove the file from the project or folder.
Import	Import the file into the project.
Source Control Operations	Source-control operations on the file.
Properties	Show the properties dialog with the file node selected.

Source navigator window

One of the best ways to find your way around your source code is using the **Source Navigator**. It parses the active project's source code and organizes classes, functions, and variables in various ways.

To activate the Source Navigator:

Choose **View** > **Source Navigator** or press **Ctrl+Alt+N**.

The main part of the **Source Navigator** window provides an overview of your application's functions, classes, and variables.

SEGGER Embedded Studio displays these icons to the left of each object:

Icon	Description
{ }	A C or C++ structure or a C++ namespace.
	A C++ class.
	A C++ member function declared `private` or a function declared with `static` linkage.
	A C++ member function declared `protected`.
	A C++ member function declared `public` or a function declared with `extern` linkage.
	A C++ member variable declared `private` or a variable declared with `static` linkage.
	A C++ member variable declared `protected`.
	A C++ member variable declared `public` or a variable declared with `extern` linkage.

Re-parsing after editing

The **Source Navigator** does not update automatically, only when you ask it to. To parse source files manually, click the **Refresh** button on the **Source Navigator** toolbar.

SEGGER Embedded Studio re-parses all files in the active project, and any dependent project, and updates the **Source Navigator** with the changes. Parsing progress is shown as a progress bar in the in the **Source Navigator** window. Errors and warnings detected during parsing are sent to the Source Navigator Log in the **Output** windowyou can show the log quickly by clicking the **Show Source Navigator Log** tool button on the **Source Navigator** toolbar.

Setting indexing threads

You can configure how many threads SEGGER Embedded Studio launches to index your project.

To set the number of threads launched when indexing a project:

Choose **View** > **Source Navigator** or press **Ctrl+Alt+N**.
Click the **Options** dropdown button at the right of the toolbar.
Move the slider to select the number of threads to launch.

Increasing the number of threads will complete indexing faster, but may reduce the responsiveness of SEGGER Embedded Studio when editing, for example. You should choose a setting that you are comfortable with for your PC. By default, SEGGER Embedded Studio launches 16 threads to index the project and is a good compromise for a desktop quad-core PC.

Sorting and grouping

You can group objects by their type; that is, whether they are classes, functions, namespaces, structures, or variables. Each object is placed into a folder according to its type.

To group objects by type:

1. On the **Source Navigator** toolbar, click the arrow to the right of the **Cycle Grouping** button.
2. Choose **Group By Type**

References window

The **References** window shows the results of the last **Find References** operation. The **Find References** facility is closely related to the **Source Navigator** in that it indexes your project and searches for references within the active source code regions.

To activate the References window:

If you have hidden the **References** window and want to see it again:

Choose **View** > **References** or press **Ctrl+Alt+R**.

To find all references in a project:

1. Open a source file that is part of the active project, or one of its dependent projects.
2. In the editor, move the insertion point within the name of the function, variable, method, or macro to find.
3. Choose **Navigate** > **Find References** or press **Alt+R**.
4. SEGGER Embedded Studio shows the **References** window, without moving focus, and searches your project in the background.

You can also find references directly from the text editor's context menu: right-click the item to find and choose **Find References**. As a convenience, SEGGER Embedded Studio is configured to also run **Find References** when you Alt+Right-click in the text editorsee **Mouse-click accelerators**.

To search within the results:

Type the text to search for in the Reference window's search box. As you type, the search results are narrowed.
Click the close button to clear the search text and show all references.

To set the number of threads launched when finding references:

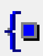
Choose **View** > **References** or press **Ctrl+Alt+R**.
Click the **Options** dropdown button at the right of the toolbar.
Move the slider to select the number of threads to launch.

Increasing the number of threads will complete searches faster, but may reduce the responsiveness of SEGGER Embedded Studio when editing, for example. You should choose a setting that you are comfortable with for your PC. By default, SEGGER Embedded Studio launches 16 threads to search the project and is a good compromise for a desktop quad-core PC.

Symbol browser window

The **Symbol Browser** shows useful information about your linked application and complements the information displayed in the **Project Explorer** window. You can select different ways to filter and group the information in the **Symbol Browser** to provide an at-a-glance overview of your application. You can use the **Symbol Browser** to *drill down* to see the size and location of each part of your program. The way symbols are sorted and grouped is saved between runs; so, when you rebuild an application, SEGGER Embedded Studio automatically updates the **Symbol Browser** so you can see the effect of your changes on the memory layout of your program.

User interface

Button	Description
	Group symbols by source filename.
	Group symbols by symbol type (equates, functions, labels, sections, and variables).
	Group symbols by the section where they are defined.
	Move the insertion point to the statement that defined the symbol.
	Select columns to display.

The main part of the **Symbol Browser** displays each symbol (both external and static) that is linked into an application. SEGGER Embedded Studio displays the following icons to the left of each symbol:

Icon	Description
	*Private Equate* A private symbol not defined relative to a section.
	*Public Equate* A public symbol that is not defined relative to a section.
	*Private Function* A private function symbol.
	*Public Function* A public function symbol.
	*Private Label* A private data symbol, defined relative to a section.
	*Public Label* A public data symbol, defined relative to a section.
	*Section* A program section.

Choosing what to show

To activate the Symbol Browser window:

Choose **View** > **Symbol Browser** or press **Ctrl+Alt+Y**.

You can choose to display the following fields for each symbol:

Value:The value of the symbol. For labels, code, and data symbols, this will be the address of the symbol. For absolute or symbolic equates, this will be the value of the symbol.
Range:The range of addresses the code or data item covers. For code symbols that correspond to high-level functions, the range is the range of addresses used for that function's code. For data addresses that correspond to high-level **static** or **extern** variables, the range is the range of addresses used to store that data item. These ranges are only available if the corresponding source file was compiled with debugging information turned on: if no debugging information is available, the range will simply be the first address of the function or data item.
Size:The size, in bytes, of the code or data item. The **Size** column is derived from the **Range** of the symbol: if the symbol corresponds to a high-level code or data item and has a range, **Size** is calculated as the difference between the start and end addresses of the range. If a symbol has no range, the size column is blank.
Section:The section in which the symbol is defined. If the symbol is not defined within a section, the **Section** column is blank.
Type:The high-level type for the data or code item. If the source file that defines the symbol is compiled with debugging information turned off, type information is not available and the **Type** column is blank.

Initially the **Range** and **Size** columns are shown in the **Symbol Browser**. To select which columns to display, use the **Field Chooser** button on the **Symbol Browser** toolbar.

To select the fields to display:

1. Click the **Field Chooser** button on the **Symbol Browser** toolbar.
2. Select the fields you wish to display and deselect the fields you wish to hide.

Organizing and sorting symbols

When you group symbols by section, each symbol is grouped underneath the section in which it is defined. Symbols that are absolute or are not defined within a section are grouped beneath (No Section).

To group symbols by section:

1. On the **Symbol Browser** toolbar, click the arrow next to the **Cycle Grouping** button.
2. From the pop-up menu, choose **Group By Section**.

The **Cycle Grouping** icon will change to indicate that the **Symbol Browser** is grouping symbols by section.

When you group symbols by type, each symbol is classified as one of the following:

An *Equate* has an absolute value and is not defined as relative to, or inside, a section.
A *Function* is defined by a high-level code sequence.
A *Variable* is defined by a high-level data declaration.
A *Label* is defined by an assembly language module. *Label* is also used when high-level modules are compiled with debugging information turned off.

When you group symbols by source file, each symbol is grouped underneath the source file in which it is defined. Symbols that are absolute, are not defined within a source file, or are compiled without debugging information, are grouped beneath (Unknown).

To group symbols by type:

1. On the **Symbol Browser** toolbar, click the arrow next to the **Cycle Grouping** button.
2. Choose **Group By Type** from the pop-up menu.

The **Cycle Grouping** icon will change to indicate that the **Symbol Browser** is grouping symbols by type.

To group symbols by source file:

1. On the **Symbol Browser** toolbar, click the arrow next to the **Cycle Grouping** button.
2. Choose **Group By Source File**.

The **Cycle Grouping** icon will change to indicate that the **Symbol Browser** is grouping symbols by source file.

When you sort symbols alphabetically, all symbols are displayed in a single list in alphabetical order.

To list symbols alphabetically:

1. On the **Symbol Browser** toolbar, click the arrow next to the **Cycle Grouping** button.
2. Choose **Sort Alphabetically**.

The **Cycle Grouping** icon will change to indicate that the **Symbol Browser** is grouping symbols alphabetically.

Filtering and finding symbols

When you're dealing with big projects with hundreds, or even thousands, of symbols, a way to filter those symbols in order to isolate just the ones you need is very useful. The **Symbol Browser**'s toolbar provides an editable *combo*box} you can use to specify the symbols you'd like displayed. You can type * to match a sequence of zero or more characters and ? to match exactly one character.

The symbols are filtered and redisplayed as you type into the combo box. Typing the first few characters of a symbol name is usually enough to narrow the display to the symbol you need. *Note:* the C compiler prefixes all high-level language symbols with an underscore character, so the variable `extern int u` or the function `void fn(void)` have low-level symbol names _u and `_fn`. The **Symbol Browser** uses the low-level symbol name when displaying and filtering, so you must type the leading underscore to match high-level symbols.

To display symbols that start with a common prefix:

Type the desired prefix text into the combo box, optionally followed by a "*".

For instance, to display all symbols that start with "**i2c_**", type "**i2c_**" and all matching symbols are displayedyou don't need to add a trailing "*" in this case, because it is implied.

To display symbols that end with a common suffix:

Type * into the combo box, followed by the required suffix.

For instance, to display all symbols that end in _data, type *_data and all matching symbols are displayedin this case, the leading * is required.

When you have found the symbol you're interested in and your source files have been compiled with debugging information turned on, you can jump to a symbol's definition using the **Go To Definition** button.

To jump to the definition of a symbol:

1. Select the symbol from the list of symbols.
2. On the **Symbol Browser** toolbar, click **Go To Definition**.

or

1. Right-click the symbol in the list of symbols.
2. Choose **Go To Definition** from the shortcut menu.

Watching symbols

If a symbol's range and type is known, you can add it to the most recently opened **Watch** window or **Memory** window.

To add a symbol to the Watch window:

1. In the **Symbol Browser**, right-click the symbol you wish to add to the **Watch** window.
2. On the shortcut menu, choose **Add To Watch**.

To add a symbol to the Memory window:

1. In the **Symbol Browser**, right-click the symbol you wish to add to the **Memory** window.

2. Choose **Locate Memory** from the shortcut menu.

Using size information

Here are a few common ways to use the **Symbol Browser**:

What function uses the most code space? What requires the most data space?

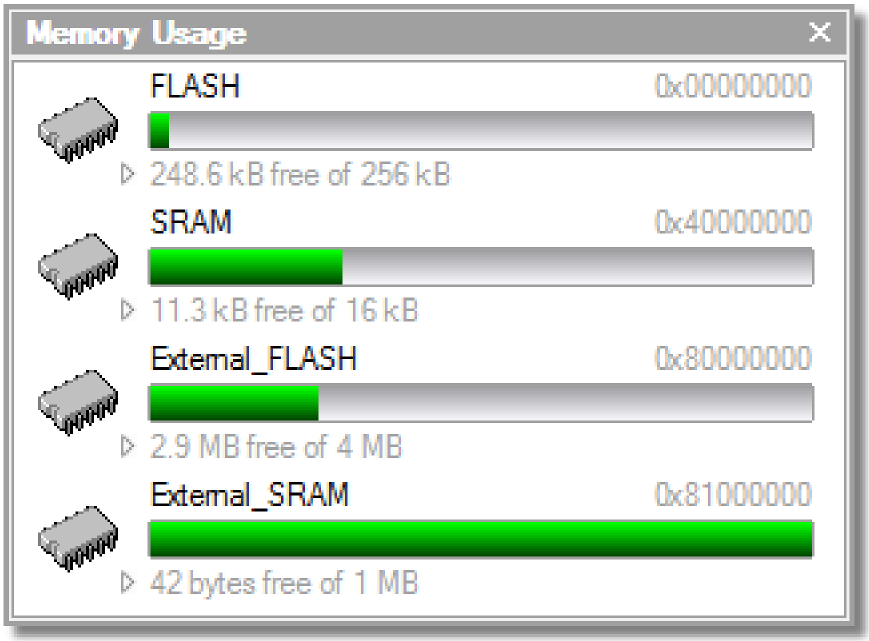
1. Choose **View** > **Symbol Browser** or press **Ctrl+Alt+Y**.
2. In the **Grouping** button menu on the **Symbol Browser** toolbar, select **Group By Type**.
3. Ensure the **Size** field is checked in the **Field Chooser** button's menu.
4. Ensure that the filter on the **Symbol Browser** toolbar is empty.
5. Click on the **Size** field in the header to sort by data size.
6. The sizes of variables and of functions are shown in separate lists.

What's the overall size of my application?

1. Choose **View** > **Symbol Browser** or press **Ctrl+Alt+Y**.
2. In the **Grouping** button menu on the **Symbol Browser** toolbar, select **Group By Section**.
3. Ensure the **Range** and **Size** fields are checked in the **Field Chooser** button's menu.
4. Read the section sizes and ranges of each section in the application.

Memory usage window

The **Memory Usage** window displays a graphical summary of how memory has been used in each memory segment of a linked application.



Each bar represents an entire memory segment. Green represents the area of the segment that contains code or data.

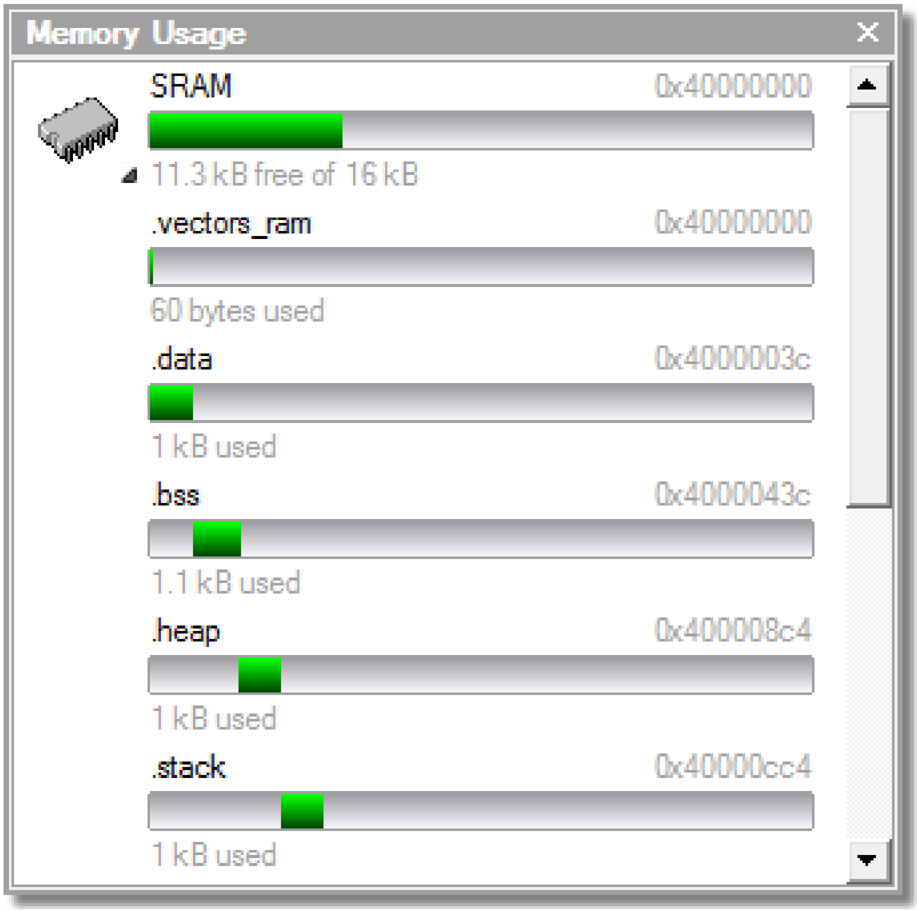
To activate the Memory Usage window:

Choose **View** > **Memory Usage** or press **Ctrl+Alt+Z**.

The memory-usage graph will only be visible if your active project's target is an executable file and the file exists. If the executable file has not been linked by SEGGER Embedded Studio, memory-usage information may not be available.

Displaying section information

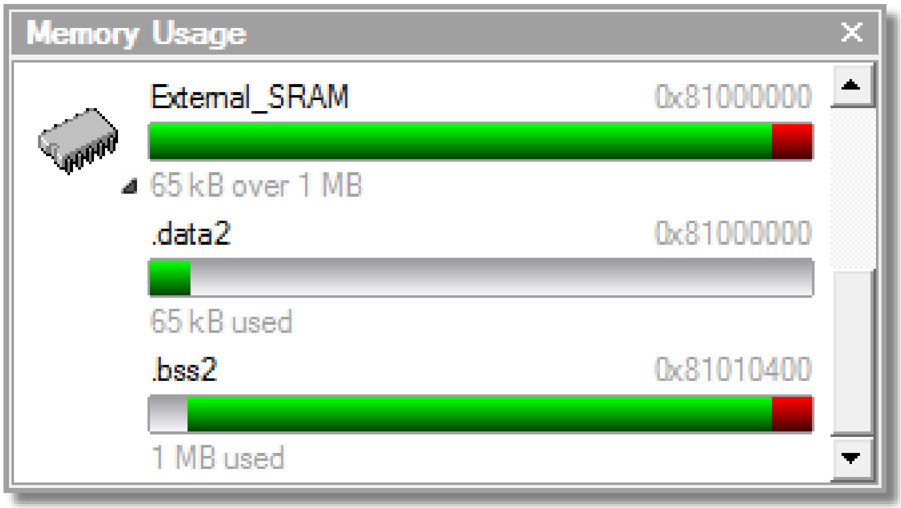
The **Memory Usage** window can also be used to visualize how program sections have been placed in memory. To display the program sections, simply click the memory segment to expand it; or, alternatively, right-click and choose **Show Memory Sections** from the shortcut menu.



Each bar represents an entire memory segment. Green represents the area of the segment that contains the program section.

Displaying segment overflow

The **Memory Usage** window also displays segment overflows when the total size of the program sections placed in a segment is larger than the segment size. When this happens, the segment and section bars represents the total memory used, green areas represent the code or data within the segment, and red areas represent code or data placed outside the segment.



Getting more-detailed information

If you require more-detailed information than that provided by the **Memory Usage** window, such as the location of specific objects within memory, use the **Symbol browser window**.

Bookmarks window

The **Bookmarks** window contains a list of bookmarks that are set in the project. The bookmarks are stored in the session file associated with the project and persist across runs of SEGGER Embedded Studioif you remove the session file, the bookmarks associated with the project are lost.

User interface

Button	Description
	Toggle a bookmark at the insertion point in the active editor. Equivalent to choosing **Edit** > **Bookmarks** > **Toggle Bookmark** or pressing **Ctrl+F2**.
	Go to the previous bookmark in the bookmark list. Equivalent to choosing **Edit** > **Bookmarks** > **Previous Bookmark** or pressing **Alt+Shift+F2**.
	Go to the next next bookmark in the bookmark list. Equivalent to choosing **Edit** > **Bookmarks** > **Next Bookmark** or pressing **Alt+F2**.
	Clear all bookmarksyou confirm the action using a dialog. Equivalent to choosing **Edit** > **Bookmarks** > **Clear All Bookmarks** or pressing **Ctrl+K, Alt+F2**.
	Selects the fill color for newly created bookmarks.

Double-clicking a bookmark in the bookmark list moves focus to the the bookmark.

You can set bookmarks with the mouse or using keystrokessee **Using bookmarks**.

Editing your code

SEGGER Embedded Studio has a built-in editor that allows you to edit text, but some features make it particularly well suited to editing code.

You can open multiple code editors to browse or edit project source code, and you can copy and paste among them. The **Windows** menu contains a list of all open code editors.

The code editor supports the language of the source file it is editing, showing code with syntax highlighting and offering smart indenting.

You can open a code editor in several ways, some of which are:

- By double-clicking a file in the **Project Explorer** or by right-clicking a file and selecting **Open** from the shortcut menu.
- Using the **File** > **New File** or **File** > **Open** commands.

Elements of the code editor

The code editor is composed of several elements, which are described here.

- *Code pane*:The area where you edit code. You can set options that affect the code pane's text indents, tabs, drag-and-drop behavior, and so forth.
- *Margin gutter*:A gray area on the left side of the code editor where margin indicators such as breakpoints, bookmarks, and shortcuts are displayed. Clicking this area sets a breakpoint on the corresponding line of code.
- *Horizontal and vertical scroll bars*:You can scroll the code pane horizontally and vertically to view code that extends beyond the edges of the pane.

Basic editing

This section is a whirlwind tour of the basic editing features SEGGER Embedded Studio's code editor provides.

Whether you are editing code, HTML, or plain text, the code editor is just like many other text editors or word processors. For code that is part of a project, the project's programming language support provides syntax highlighting (colorization), indentation, and so on.

This section *is not* a reference for everything the code editor provides; for that, look in the following sections.

Moving the insertion point

The most common way to navigate through text is to use use the mouse or the keyboard's cursor keys.

Using the mouse

You can move the insertion point within a document by clicking the mouse inside the editor window.

Using the keyboard

The keystrokes most commonly used to navigate through a document are:

Keystroke	Description
Up	Move the insertion point up one line
Down	Move the insertion point down one line
Left	Move the insertion point left one character
Right	Move the insertion point right one character
Home	Move the insertion point to the first non-whitespace character on the line pressing **Home** a second time moves the insertion point to the leftmost column
End	Move the insertion point to the end of the line
PageUp	Move the insertion point up one page
PageDown	Move the insertion point down one page
Ctrl+Home	Move the insertion point to the start of the document
Ctrl+End	Move the insertion point to the end of the document
Ctrl+Left	Move the insertion point left one word
Ctrl+Right	Move the insertion point right one word

SEGGER Embedded Studio offers additional movement keystrokes, though most users are more comfortable using repeated simple keystrokes to accomplish the same thing:

Keystroke	Description
Alt+Up	Move the insertion point up five lines
Alt+Down	Move the insertion point down five lines
Alt+Home	Move the insertion point to the top of the window
Alt+End	Move the insertion point to the bottom of the window
Ctrl+Up	Scroll the document up one line in the window without moving the insertion point

Ctrl+Down	Scroll the document down one line in the window without moving the insertion point

If you are editing source code, the are source-related keystrokes too:

Keystroke	Description
Ctrl+PgUp	Move the insertion point backwards to the previous function or method.
Ctrl+PgDn	Move the insertion point forwards to the next function or method.

Adding text

The editor has two text-input modes:

> *Insertion mode*:As you type on the keyboard, text is entered at the insertion point and any text to the right of the insertion point is shifted along. A visual indication of insertion mode is that the cursor is a flashing line.
> *Overstrike mode*:As you type on the keyboard, text at the insertion point is replaced with your typing. A visual indication of insertion mode is that the cursor is a flashing block.

Insert and overstrike modes are common to *all* editors: if one editor is in insert mode, *all* editors are in insert mode. To configure the cursor appearance, choose **Tools** > **Options**.

To toggle between insertion and overstrike mode:

> Click **Insert**.

When overstrike mode is enabled, the mode indicator changes from **INS** to **OVR** and the cursor will change to the overstrike cursor.

To add or insert text:

1. Move the insertion point to the place text is to be inserted.
2. Enter the text using the keyboard.

To overwrite characters in an existing line, press the **Insert** key to place the editor into overstrike mode.

Deleting text

The text editor supports the following common editing keystrokes:

Keystroke	Description
Backspace	Delete the character before the insertion point
Delete	Delete the character after the insertion point
Ctrl+Backspace	Delete one word before the insertion point
Ctrl+Delete	Delete one word after the insertion point

To delete characters or words:

1. Place the insertion point before the word or letter you want to delete.
2. Press **Delete** as many times as needed.

or

1. Place the insertion point after the letter or word you want to delete.
2. Press **Backspace** as many times as needed.

To delete text that spans more than a few characters:

1. Select the text you want to delete.
2. Press **Delete** or **Backspace** to delete it.

Using the clipboard

You can select text by using the keyboard or the mouse.

To select text with the keyboard:

Hold down the **Shift** key while using the cursor keys.

To select text with the mouse:

1. Click the start of the selection.
2. Drag the mouse to mark the selection.
3. Release the mouse to end selecting.

To copy selected text to the clipboard:

Choose **Edit** > **Copy** or press **Ctrl+C**.

The standard Windows key sequence **Ctrl+Ins** also copies text to the clipboard.

To cut selected text to the clipboard:

Choose **Edit** > **Cut** or press **Ctrl+X**.

The standard Windows key sequence **Shift+Del** also cuts text to the clipboard.

To insert the clipboard content at the insertion point:

Choose **Edit** > **Paste** or press **Ctrl+V**.

The standard Windows key sequence **Shift+Ins** also inserts the clipboard content at the insertion point.

Undo and redo

The editor has an *undo* facility to undo previous editing actions. The *redo* feature can be used to re-apply previously undone actions.

To undo one editing action:

Choose **Edit** > **Undo** or press **Ctrl+Z**.

The standard Windows key sequence **Alt+Backspace** also undoes an edit.

To undo multiple editing actions:

1. On the **Standard** toolbar, click the arrow next to the **Undo** button.
2. Select the editing operations to undo.

To undo all edits:

Choose **Edit** > **Others** > **Undo All** or press **Ctrl+K, Ctrl+Z**.

To redo one editing action:

Choose **Edit** > **Redo** or press **Ctrl+Y**.

The standard Windows key sequence **Alt+Shift+Backspace** also redoes an edit.

To redo multiple editing actions:

1. On the **Standard** toolbar, click the arrow next to the **Redo** tool button.
2. From the pop-up menu, select the editing operations to redo.

To redo all edits:

Choose **Edit** > **Others** > **Redo All** or press **Ctrl+K, Ctrl+Y**.

Drag and drop

You can select text, then drag it to another location. You can drop the text at a different location in the same window or in another one.

To drag and drop text:

1. Select the text you want to move.
2. Press and hold the mouse button to drag the selected text to where you want to place it.
3. Release the mouse button to drop the text.

Dragging text *moves* it to the new location. To *copy* it to a new location, hold down the **Ctrl** key while dragging the text: the mouse pointer changes to indicate a copy operation. Press the **Esc** key while dragging text to cancel the drag-and-drop edit.

By default, drag-and drop-editing is *disabled* and you must enable it if you want to use it.

To enable or disable drag-and-drop editing:

1. Choose **Tools** > **Options** or press **Alt+,**.
2. Click **Text Editor**.
3. Set **Allow Drag and Drop Editing** to **Yes** to enable or to **No** to disable drag-and-drop editing.

Searching

To find text in the current file:

1. Press **Ctrl+F**.
2. Enter the string to search for.

As you type, the editor searches the file for a match. The pop-up shows how many matches are in the current file. To move through the matches while the **Find** box is still active, press **Tab** or **F3** to move to the next match and **Shift+Tab** or **Shift+F3** to move to the previous match.

If you press **Ctrl+F** a second time, SEGGER Embedded Studio pops up the standard **Find** dialog to search the file. If you wish to bring up the **Find** dialog without pressing `Ctrl+F` twice, choose **Search** > **Find**.

Advanced editing

You can do anything using its basic code-editing features, but the SEGGER Embedded Studio text editor has a host of labor-saving features that make editing programs a snap.

This section describes the code-editor features intended to make editing source code easier.

Indenting source code

The editor uses the **Tab** key to increase or decrease the indentation level of the selected text.

To increase indentation:

Select the text to indent.
Choose **Selection** > **Increase Line Indent** or press **Tab**.

To decrease indentation:

Select the text to indent.
Choose **Selection** > **Decrease Line Indent** or press **Shift+Tab**.

The indentation size can be changed in the **Language Properties** pane of the editor's **Properties** window, as can all the indent-related features listed below.

To change indentation size:

Choose **Tools** > **Options** or press **Alt+,**.
Select the **Languages** page.
Set the **Indent Size** environment option for the required language.

You can choose to use spaces or tab tab characters to fill whitespace when indenting.

To set tab or space fill when indenting:

Choose **Tools** > **Options** or press **Alt+,**.
Select the **Languages** page.
Set the **Use Tabs** environment option for the required language. *Note:* changing this setting does not add or remove existing tabs from files, the change will only affect new indents.

The editor can assist with source code indentation while inserting text. There are three levels of indentation assistance:

None:The indentation of the source code is left to the user.
Indent:This is the default. The editor maintains the current indentation level. When you press **Return** or **Enter**, the editor moves the insertion point down one line and indented to the same level as the now-previous line.
Smart:The editor analyzes the source code to compute the appropriate indentation level for each line. You can change how many lines before the insertion point will be analyzed for context. The smart-indent mode can be configured to indent either open and closing braces or the lines following the braces.

Changing indentation options:

To change the indentation mode:

Set the **Indent Mode** environment option for the required language.

To change whether opening braces are indented in smart-indent mode:

Set the **Indent Opening Brace** environment option for the required language.

To change whether closing braces are indented in smart-indent mode:

Set the **Indent Closing Brace** environment option for the required language.

To change the number of previous lines used for context in smart-indent mode:

Set the **Indent Context Lines** environment option for the required language.

Commenting out sections of code

To comment selected text:

Choose **Selection** > **Comment** or press **Ctrl**+/.

To uncomment selected text:

Choose **Selection** > **Uncomment** or press **Ctrl+Shift**+/.

You can also toggle the commenting of a selection by typing **/**. This has no menu equivalent.

Adjusting letter case

The editor can change the case of the current word or the selection. The editor will change the case of the selection, if there is a selection, otherwise it will change the case of word at the insertion point.

To change text to uppercase:

Choose **Selection** > **Make Uppercase** or press **Ctrl**+**Shift**+**U**.

This changes, for instance, Hello to HELLO.

To change text to lowercase:

Choose **Selection** > **Make Lowercase** or press **Ctrl**+**U**.

This changes, for instance, Hello to hello.

To switch between uppercase and lowercase:

Choose **Selection** > **Switch Case**.

This changes, for instance, Hello to hELLO.

With large software teams or imported source code, sometimes identifiers don't conform to your local coding style. To assist in conversion between two common coding styles for identifiers, SEGGER Embedded Studio's editor offers the following two shortcuts:

To change from split case to camel case:

Choose **Selection** > **Camel Case** or press **Ctrl**+**K, Ctrl**+**Shift**+**U**.

This changes, for instance, this_is_wrong to thisIsWrong.

To change from camel case to split case:

Choose **Selection** > **Split Case** or press **Ctrl**+**K, Ctrl**+**U**.

This changes, for instance, thisIsWrong to this_is_wrong.

Using bookmarks

To edit a document elsewhere and then return to your current location, add a bookmark. The **Bookmarks** window maintains a list of the bookmarks set in source files see **Bookmarks window**.

To place a bookmark:

1. Move the insertion point to the line you wish to bookmark.
2. Choose **Edit** > **Bookmarks** > **Toggle Bookmark** or press **Ctrl+F2**.

A bookmark symbol appears next to the line in the indicator margin to show the bookmark is set.

To place a bookmark using the mouse:

1. Right-click the margin gutter where the bookmark should be set.
2. Choose **Toggle Bookmark**.

The default color to use for new bookmarks is configured in the **Bookmarks** window. You can choose a specific color for the bookmark as follows:

1. Press and hold the **Alt** key.
2. Click the margin gutter where the bookmark should be set.
3. From the palette, click the bookmark color to use for the bookmark.

To navigate forward through bookmarks:

1. Choose **Edit** > **Bookmarks** > **Next Bookmark In Document** or press **F2**.
2. The editor moves the insertion point to the next bookmark in the document.

If there is no following bookmark, the insertion point moves to the first bookmark in the document.

To navigate backward through bookmarks:

1. Choose **Edit** > **Bookmarks** > **Previous Bookmark In Document** or press **Shift+F2**.
2. The editor moves the insertion point to the previous bookmark in the document.

If there is no previous bookmark, the insertion point moves to the last bookmark in the document.

To remove a bookmark:

1. Move the insertion point to the line containing the bookmark.
2. Choose **Edit** > **Bookmarks** > **Toggle Bookmark** or press **Ctrl+F2**.

The bookmark symbol disappears, indicating the bookmark is no longer set.

To remove all bookmarks in a document:

Choose **Edit** > **Bookmarks** > **Clear Bookmarks In Document** or press **Ctrl+K, F2**.

Quick reference for bookmark operations

Keystroke	Menu	Description
Ctrl+F2	**Edit > Bookmarks > Toggle Bookmark**	Toggle a bookmark at the insertion point.
Ctrl+K, 0		Clear the bookmark at the insertion point.
F2	**Edit > Bookmarks > Next Bookmark In Document**	Move the insertion point to next bookmark in the document.
Shift+F2	**Edit > Bookmarks > Previous Bookmark In Document**	Move the insertion point to previous bookmark in the document.
Ctrl+Q, F2	**Edit > Bookmarks > First Bookmark In Document**	Move the insertion point to the first bookmark in the document.
Ctrl+Q, Shift+F2	**Edit > Bookmarks > Last Bookmark In Document**	Move the insertion point to the last bookmark in the document.
Ctrl+K, F2	**Edit > Bookmarks > Clear Bookmarks In Document**	Clear all bookmarks in the document.
Alt+F2	**Edit > Bookmarks > Next Bookmark**	Move the insertion point to the next bookmark in the Bookmarks list.
Alt+Shift+F2	**Edit > Bookmarks > Previous Bookmark**	Move the insertion point to the previous bookmark in the Bookmarks list.
Ctrl+Q, Alt+F2	**Edit > Bookmarks > First Bookmark**	Move the insertion point to the first bookmark in the Bookmarks list.
Ctrl+Q, Alt+Shift+F2	**Edit > Bookmarks > Last Bookmark**	Move the insertion point to the last bookmark in the Bookmarks list.
Ctrl+K, Alt+F2	**Edit > Bookmarks > Clear All Bookmarks**	Clear all bookmarks in all documents.

Find and Replace window

The **Find and Replace** window allows you to search for and replace text in the current document or in a range of specified files.

To activate the Find and Replace window:

Choose **Search** > **Replace in Files** or press **Ctrl+Alt+F**.

To find text in a single file:

Select **Current Document** in the context combo box.
Enter the string to be found in the text edit input.
If the search will be case sensitive, set the **Match case** option.
If the search will be for a whole wordi.e., there will be whitespace, such as spaces or the beginning or end of the line, on both sides of the string being searched forset the **Whole word** option.
If the search string is a regular expression, set the **Use regexp** option.
Click the **Find** button to find all occurrences of the string in the current document.

To find and replace text in a single file:

Click the **Replace** button on the toolbar.
Enter the string to search for into the **Find what** input.
Enter the replacement string into the **Replace with** input. If the search string is a regular expression, the *n* back-reference can be used in the replacement string to reference captured text.
If the search will be case sensitive, set the **Match case** option.
If the search will be for a whole wordi.e., there will be whitespace, such as spaces or the beginning or end of the line, on both sides of the string being searched forset the **Match whole word** option.
If the search string is a regular expression, set the **Use regular expression** option.
Click the **Find Next** button to find next occurrence of the string, then click the **Replace** button to replace the found string with the replacement string; or click **Replace All** to replace all occurrences of the search string without prompting.

To find text in multiple files:

Click the **Find In Files** button on the toolbar.
Enter the string to search for into the **Find what** input.
Select the appropriate option in the **Look in** input to select whether to carry out the search in all open documents, all documents in the current project, all documents in the current solution, or all files in a specified folder.
If you have specified that you want to search in a folder, select the folder you want to search by entering its path in the **Folder** input and use the **Look in files matching** input to specify the type of files you want to search.

If the search will be case sensitive, set the **Match case** option.
If the search will be for a whole wordi.e., there will be whitespace, such as spaces or the beginning or end of the line, on both sides of the string being searched forset the **Match whole word** option.
If the search string is a regular expression, set the **Use regular expression** option.
Click the **Find All** button to find all occurrences of the string in the specified files, or click the **Bookmark All** button to bookmark all the occurrences of the string in the specified files.

To replace text in multiple files:

Click the **Replace In Files** button on the toolbar.
Enter the string to search for into the **Find what** input.
Enter the replacement string into the **Replace with** input. If the search string is a regular expression, the *n* back-reference can be used in the replacement string to reference captured text.
Select the appropriate option in the **Look in** input to select whether you want to carry out the search and replace in all open documents, all documents contained in the current project, all documents in the current solution, or all files in a specified folder.
If you have specified that you want to search in a folder, select the folder you want to search by entering its path in the **Folder** input and use the **Look in files matching** input to specify the type of files you want to search.
If the search will be case sensitive, set the **Match case** option.
If the search will be for a whole wordi.e., there will be whitespace, such as spaces or the beginning or end of the line, on both sides of the string being searched forset the **Match whole word** option.
If the search string is a regular expression, set the **Use regular expression** option.
Click the **Replace All** button to replace all occurrences of the string in the specified files.

Clipboard Ring window

The code editor captures all cut and copy operations, and stores the cut or copied item on the *clipboard ring*. The clipboard ring stores the last 20 cut or copied text items, but you can configure the maximum number by using the environment options dialog. The clipboard ring is an excellent place to store scraps of text when you're working with many documents and need to cut and paste between them.

To activate the clipboard ring:

Choose **Edit** > **Clipboard Ring** > **Clipboard Ring** or press **Ctrl+Alt+C**.

To paste from the clipboard ring:

1. Cut or copy some text from your code. The last item you cut or copy into the clipboard ring is the current item for pasting.
2. Press **Ctrl+Shift+V** to paste the clipboard ring's current item into the current document.
3. Repeatedly press **Ctrl+Shift+V** to cycle through the entries in the clipboard ring until you get to the one you want to permanently paste into the document. Each time you press **Ctrl+Shift+V**, the editor replaces the last entry you pasted from the clipboard ring, so you end up with just the last one you selected. The item you stop on then becomes the current item.
4. Move to another location or cancel the selection. You can use **Ctrl+Shift+V** to paste the current item again or to cycle the clipboard ring to a new item.

Clicking an item in the clipboard ring makes it the current item.

To paste a specific item from the clipboard ring:

1. Move the insertion point to the position to paste the item in the document.
2. Click the arrow at the right of the item to paste.
3. Choose *Paste* from the pop-up menu.

or

1. Click the item to paste to make it the current item.
2. Move the insertion point to the position to paste the item in the document.
3. Press **Ctrl+Shift+V**.

To paste all items into a document:

To paste all items on the clipboard ring into the current document, move the insertion point to where you want to paste the items and do one of the following:

Choose **Edit** > **Clipboard Ring** > **Paste All**.

or

On the **Clipboard Ring** toolbar, click the **Paste All** button.

To remove an item from the clipboard ring:

1. Click the arrow at the right of the item to remove.
2. Choose **Delete** from the pop-up menu.

To remove all items from the clipboard ring:

Choose **Edit** > **Clipboard Ring** > **Clear Clipboard Ring**.

or

On the **Clipboard Ring** toolbar, click the **Clear Clipboard Ring** button.

To configure the clipboard ring:

1. Choose **Tools** > **Options** or press **Alt+,**.
2. Click the **Windows** category to show the **Clipboard Ring Options** group.
3. Select **Preserve Contents Between Runs** to save the content of the clipboard ring between runs, or deselect it to start with an empty clipboard ring.
4. Change **Maximum Items Held In Ring** to configure the maximum number of items stored on the clipboard ring.

Mouse-click accelerators

SEGGER Embedded Studio provides a number of mouse-click accelerators in the editor that speed access to commonly used functions. The mouse-click accelerators are user configurable using **Tools** > **Options**.

Default mouse-click assignments

Click	Default
Left	Not configurable start selection.
Shift+Left	Not configurable extend selection.
Ctrl+Left	Select word.
Alt+Left	Execute **Go To Definition**.
Middle	No action.
Shift+Middle	Display **Go To Include** menu.
Ctrl+Middle	No action.
Alt+Middle	Display **Go To Method** menu.
Right	Not configurable show context menu.
Shift+Right	No action.
Ctrl+Right	No action.
Alt+Right	Execute **Find References**.

Each accelerator can be assigned one of the following actions:

Default:The system default for that click.
Go To Definition:Go to the definition of the item clicked, equivalent to choosing **Navigate** > **Go To Definition** or pressing **Alt+G**.
Find References:Find references to the item clicked, equivalent to choosing **Navigate** > **Find References** or pressing **Alt+R**.
Find in Solution:Textually find the item clicked in all the files in the solution, equivalent to choosing **Search** > **Find Extras** > **Find in Solution** or pressing **Alt+U**.
Find Help:Use F1-help on the item clicked, equivalent to choosing **Help** > **Help** or pressing **F1**.
Go To Method:Display the **Go To Method** menu, equivalent to choosing **Navigate** > **Find Method** or pressing **Ctrl+M**.
Go To Include:Display the **Go To Include** menu, equivalent to choosing **Navigate** > **Find Include** or pressing **Ctrl+Shift+M**.
Paste:Paste the clipboard at the position clicked, equivalent to choosing **Edit** > **Paste** or pressing **Ctrl+V**.

Configuring Mac OS X

On Mac OS X you must configure the mouse to pass middle clicks and right clicks to the application if you wish to use mouse-click accelerators in SEGGER Embedded Studio. Configure the mouse preferences in the **Mouse** control panel in Mac OS X **System Preferences** to the following:

Right mouse button set to **Secondary Button**.

Middle mouse button set to **Button 3**.

Regular expressions

The editor can search and replace text using *regular expressions*. A regular expression is a string that uses special characters to describe and reference patterns of text. The regular expression system used by the editor is modeled on Perl's regexp language. For more information on regular expressions, see *Mastering Regular Expressions*, Jeffrey E F Freidl, ISBN 0596002890.

Summary of special characters

The following table summarizes the special characters the SEGGER Embedded Studio editor supports

Pattern	Description
\d	Match a numeric character.
\D	Match a non-numeric character.
\s	Match a whitespace character.
\S	Match a non-whitespace character.
\w	Match a word character.
\W	Match a non-word character.
[*c*]	Match set of characters; e.g., [ch] matches characters c or h. A range can be specified using the - character; e.g., [0-27-9] matches if the character is 0, 1, 2, 7 8, or 9. A range can be negated using the ^ character; e.g., [^a-z] matches if the character is anything other than a lowercase alphabetic character.
c	Match the literal character *c*. For example, you would use * to match the character *.
\a	Match ASCII bell character (ASCII code 7).
\f	Match ASCII form feed character (ASCII code 12).
\n	Match ASCII line feed character (ASCII code 10).
\r	Match ASCII carriage return character (ASCII code 13).
\t	Match ASCII horizontal tab character (ASCII code 9).
\v	Match ASCII vertical tab character.
\x*hhhh*	Match Unicode character specified by hexadecimal number *hhhh*.
.	Match any character.
*	Match zero or more occurrences of the preceding expression.
+	Match one or more occurrences of the preceding expression.

?	Match zero or one occurrences of the preceding expression.
{*n*}	Match *n* occurrences of the preceding expression.
{*n*,}	Match at least *n* occurrences of the preceding expression.
{,*m*}	Match at most *m* occurrences of the preceding expression.
{*n*,*m*}	Match at least *n* and at most *m* occurrences of the preceding expression.
^	Beginning of line.
$	End of line.
\b	Word boundary.
\B	Non-word boundary.
(*e*)	Capture expression *e*.
n	Back-reference to *n*th captured text.

Examples

The following regular expressions can be used with the editor's search-and-replace operations. To use the regular expression mode, the **Use regular expression** checkbox must be set in the search-and-replace dialog. Once enabled, regular expressions can be used in the **Find what** search string. The **Replace With** strings can use the "*n*" back-reference string to reference any captured strings.

"Find what"	"Replace With"	Description
u\w.d		Search for any-length string containing one or more word characters beginning with the character u and ending in the character d.
^.*;$		Search for any lines ending in a semicolon.
(typedef.+\s+)(\S+);	\1TEST_\2;	Find C type definition and insert the string TEST onto the beginning of the type name.

Locals window

The **Locals** window displays a list of all variables that are in scope of the selected stack frame in the **Call Stack**.

The **Locals** window has a toolbar and a main data display.

Button	Description
	Display the selected item in binary.
	Display the selected item in octal.
	Display the selected item in decimal.
	Display the selected item in hexadecimal.
	Display the selected item as a signed decimal.
	Display the selected item as a character or Unicode character.
	Set the range displayed in the active **Memory** window to span the memory allocated to the selected item.
	Sort variables alphabetically by name.
	Sort variables numerically by address or register number (default).

Using the Locals window

The **Locals** window shows the local variables of the active function when the debugger is stopped. The contents of the **Locals** window changes when you use the **Debug Location** toolbar items or select a new frame in the **Call Stack** window. When the program stops at a breakpoint, or is stepped, the **Locals** window updates to show the active stack frame. Items that have changed since they were previously displayed are highlighted in red.

To activate the Locals window:

Choose **View** > **Locals** or press **Ctrl+Alt+L**.

When you select a variable in the main part of the display, the display-format button highlighted on the **Locals** window toolbar changes to show the selected item's display format.

To change the display format of a local variable:

Right-click the item to change.
From the shortcut menu, choose the desired display format.

or

Click the item to change.

On the **Locals** window toolbar, select the desired display format.

To modify the value of a local variable:

Click the value of the local variable to modify.

Enter the new value for the local variable. Prefix hexadecimal numbers with **0x**, binary numbers with **0b**, and octal numbers with **0**.

or

Right-click the value of the local variable to modify.

From the shortcut menu, select one of the commands to modify the local variable's value.

Globals window

The **Globals** window displays a list of all variables that are global to the program. The operations available on the entries in this window are the same as the **Watch** window, except you cannot add or delete variables from the **Globals** window.

Globals window user interface

The **Globals** window consists of a toolbar and main data display.

Globals toolbar

Button	Description
	Display the selected item in binary.
	Display the selected item in octal.
	Display the selected item in decimal.
	Display the selected item in hexadecimal.
	Display the selected item as a signed decimal.
	Display the selected item as a character or Unicode character.
	Set the range displayed in the active **Memory** window to span the memory allocated to the selected item.
	Sort variables alphabetically by name.
	Sort variables numerically by address or register number (default).

Using the Globals window

The **Globals** window shows the global variables of the application when the debugger is stopped. When the program stops at a breakpoint, or is stepped, the **Globals** window updates to show the active stack frame and new variable values. Items that have changed since they were previously displayed are highlighted in red.

To activate the Globals window:

Choose **View** > **Globals** or press **Ctrl+Alt+G**.

Changing the display format

When you select a variable in the main part of the display, the display-format button highlighted on the **Globals** window toolbar changes to show the item's display format.

To change the display format of a global variable:

Right-click the item to change.
From the shortcut menu, choose the desired display format.

or

Click the item to change.
On the **Globals** window toolbar, select the desired display format.

To modify the value of a global variable:

Click the value of the global variable to modify.
Enter the new value for the global variable. Prefix hexadecimal numbers with **0x**, binary numbers with **0b**, and octal numbers with **0**.

Watch window

The **Watch** window provides a means to evaluate expressions and to display the results of those expressions. Typically, expressions are just the name of a variable to be displayed, but they can be considerably more complex; see **Debug expressions**. *Note:* expressions are always evaluated when your program stops, so the expression you are watching is the one that is in scope of the stopped program position.

The **Watch** window is divided into a toolbar and the main data display.

Button	Description
	Display the selected item in binary.
	Display the selected item in octal.
	Display the selected item in decimal.
	Display the selected item in hexadecimal.
	Display the selected item as a signed decimal.
	Display the selected item as a character or Unicode character.
	Set the range displayed in the active **Memory** window to span the memory allocated to the selected item.
	Remove the selected watch item.
	Remove all the watches.

Right-clicking a watch item shows a shortcut menu with commands that are not available from the toolbar.

Button	Description
	View pointer or array as a null-terminated string.
	View pointer or array as an array.
	View pointer value.
	Set watch value to zero.
	Set watch value to one.
	Increment watched variable by one.
	Decrement watched variable by one.

	Negated watched variable.
	Invert watched variable.
	View the properties of the watch value.

You can view details of the watched item using the **Properties** dialog.

Filename
The filename context of the watch item.

Line number
The line number context of the watch item.

(Name)
The name of the watch item.

Address
The address or register of the watch item.

Expression
The debug expression of the watch item.

Previous Value
The previous watch value.

Size In Bytes
The size of the watch item in bytes.

Type
The type of the watch item.

Value
The value of the watch item.

Using the Watch window

Each expression appears as a row in the display. Each row contains the expression and its value. If the value of an expression is structured (for example, an array), you can open the structure to see its contents.

The display updates each time the debugger locates to source code. So it will update each time your program stops on a breakpoint, or single steps, and whenever you traverse the call stack. Items that have changed since they were previously displayed are highlighted in red.

To activate the Watch window:

Choose **View** > **Watch** > **Watch 1** or press **Ctrl+T, W, 1**.

You can show other **Watch** windows similarly.

You can add a new expression to be watched by clicking and typing into the last entry in the **Watch** window. You can change an expression by clicking its entry and editing its contents.

When you select a variable in the main part of the display, the display format button highlighted on the **Watch** window toolbar changes to show the item's display format.

To change the display format of an expression:

Right-click the item to change.
From the shortcut menu, choose the desired display format.

or

Click the item to change.
On the **Watch** window toolbar, select the desired display format.

The selected display format will then be used for all subsequent displays and will be preserved after the debug session stops.

For C programs, the interpretation of pointer types can be changed by right-clicking and selecting from the shortcut menu. A pointer can be interpreted as:

a null-terminated ASCII string
an array
an integer
dereferenced

To modify the value of an expression:

Click the value of the local variable to modify.
Enter the new value of the local variable. Prefix hexadecimal numbers with **0x**, binary numbers with **0b**, and octal numbers with **0**.

or

Right-click the value of the local variable to modify.
From the shortcut menu, choose one of the commands to modify the variable's value.

Register window

The **Register** windows show the values of both CPU registers and the processor's special function or peripheral registers. Because microcontrollers are becoming very highly integrated, it's not unusual for them to have hundreds of special function registers or peripheral registers, so SEGGER Embedded Studio provides four register windows. You can configure each register window to display one or more register groups for the processor being debugged.

A **Register** window has a toolbar and a main data display.

Button	Description
	Display the CPU, special function register, and peripheral register groups.
	Display the CPU registers.
	Hide the CPU registers.
	Force-read a register, ignoring the access attribute of the register.
	Update the selected register group.
	Set the active memory window to the address and size of the selected register group.

Using the registers window

Both CPU registers and special function registers are shown in the main part of the **Registers** window. When the program stops at a breakpoint, or is stepped, the **Registers** windows update to show the current values of the registers. Items that have changed since they were previously displayed are highlighted in red.

To activate the first register window:

Choose **View** > **Registers** > **Registers 1** or press **Ctrl+T, R, 1**.

Other register windows can be similarly activated.

Displaying CPU registers

The values of the CPU registers displayed in the **Registers** window depend up upon the selected context. The selected context can be:

The register state the CPU stopped in.
The register state when a function call occurred using the Call Stack window.

The register state of the currently selected thread using the the **Threads** window.
The register state you supplied with the **Debug** > **Locate** operation.

To display a group of CPU registers:

On the **Registers** window toolbar, click the **Groups** button.
From the pop-up menu, select the register groups to display and deselect the ones to hide.

You can deselect all CPU register groups to allow more space in the display for special function registers or peripheral registers. So, for instance, you can have one register window showing the CPU registers and other register windows showing different peripheral registers.

Displaying special function or peripheral registers

The **Registers** window shows the set of register groups defined in the memory-map file the application was built with. If there is no memory-map file associated with a project, the **Registers** window will show only the CPU registers.

To display a special function or peripheral register:

On the **Registers** toolbar, click the **Groups** button.
From the pop-up menu, select the register groups to display and deselect the ones to hide.

Changing display format

When you select a register in the main part of the display, the display-format button highlighted on the **Registers** window toolbar changes to show the item's display format.

To change the display format of a register:

Right-click the item to change.
From the shortcut menu, choose the desired display format.

or

Click the item to change.
On the **Registers** window toolbar, select the desired display format.

Modifying register values

To modify the value of a register:

Click the value of the register to modify.

Enter the new value for the register. Prefix hexadecimal numbers with **0x**, binary numbers with **0b**, and octal numbers with **0**.

or

Right-click the value of the register to modify.
From the shortcut menu, choose one of the commands to modify the register value.

Modifying the saved register value of a function or thread may not be supported.

Memory window

The **Memory** window shows the contents of the connected target's memory areas and allows the memory to be edited. SEGGER Embedded Studio provides four memory windows, you can configure each memory window to display different memory ranges.

The **Memory** window has a toolbar and a data display/edit area

Field/Button	Description
Address	Address to display. This can be a numeric value or a debug expression.
Size	Number of bytes to display. This can be a number or a debug expression. If unspecified, the number of bytes required to fill the window will be automatically calculated.
Columns	Number of columns to display. If unspecified, the number of columns required to fill the window will be automatically calculated.
	Select binary display.
	Select octal display.
	Select unsigned decimal display.
	Select signed decimal display.
	Select hexadecimal display *(default)*.
	Select byte display *(default)*.
	Select 2-byte display.
	Select 4-byte display.
	Display both data and text *(default)*.
	Display data only.
	Display text only.
	Display an incrementing address range that starts from the selected address *(default)*.
	Display a decrementing address range that starts from the selected address.

	Display an incrementing address range that ends at the selected address.
	Display a decrementing address range that ends at the selected address.
	Evaluate the address and size expressions, and update the **Memory** window.

Using the memory window

The memory window does not show the complete address space of the target, instead you must enter both the address and the number of bytes to display. You can specify the address and size using numeric values or **debug expressions** which enable you to position the memory display at the address of a variable or at the value of a register. You can also specify whether you want the expressions to be evaluated each time the memory window is updated, or you can re-evaluate them yourself with the press of a button. Memory windows update each time your program stops on a breakpoint, after a single step and whenever you traverse the call stack. If any values that were previously displayed have changed, they are highlighted in red.

To activate the first Memory window:

Choose **View** > **Memory** > **Memory 1** or press **Ctrl+T, M, 1**.

Other register windows can be similarly activated.

Using the mouse

You can move the memory window's edit cursor by clicking on a data or text entry.

The vertical scroll bar can be used to modify the address being viewed by clicking the up and down buttons, the page up and down areas or using the vertical scroll wheel when the scroll bar is at it's furthest extent.

Using the keyboard

Keystroke	Description
Up	Move the cursor up one line, or if the cursor is on the first line, move the address up one line.
Down	Move the cursor down one line, or if the cursor is on the last line, move the address down line line.
Left	Move the cursor left one character.
Right	Move the cursor right one character.
Home	Move the cursor to the first entry.
End	Move the cursor to the last entry.
PageUp	Move the cursor up one page, or if the cursor is on first page, move the address up one page.

PageDown	Move the cursor down one page, or if the cursor is on the last page, move the address down one page.
Ctrl+E	Toggle the cursor between data and text editing.

Editing memory

To edit memory, simply move the cursor to the data or text entry you want to modify and start typing. The memory entry will be written and read back as you type.

Shortcut menu commands

The shortcut menu contains the following commands:

Action	Description
Access Memory By Display Width	Access memory in terms of the display width.
Address Order	Specify whether the address range shown uses **Address** as the start or end address and whether addresses should increment or decrement.
Auto Evaluate	Re-evaluate **Address** and **Size** each time the **Memory** window is updated.
Auto Refresh	Specify how frequently the memory window should automatically refresh.
Export To Binary Editor	Create a binary editor with the current **Memory** window contents.
Save As	Save the current **Memory** window contents to a file. Supported file formats are **Binary File**, **Motorola S-Record File**, **Intel Hex File**, **TI Hex File**, and **Hex File**.
Load From	Load the current **Memory** window from a file. Supported file formats are **Binary File**, **Motorola S-Record File**, **Intel Hex File**, **TI Hex File**, and **Hex File**.

Display formats

You can set the **Memory** window to display 8-bit, 16-bit, and 32-bit values that are formatted as hexadecimal, decimal, unsigned decimal, octal, or binary. You can also specify how many columns to display.

Saving memory contents

You can save the displayed contents of the memory window to a file in various formats. Alternatively, you can export the contents to a binary editor to work on them.

You can save the displayed memory values as a binary file, Motorola S-record file, Intel hex file, or a Texas Instruments TXT file.

To save the current state of memory to a file:

Select the start address and number of bytes to save by editing the **Start Address** and **Size** fields in the **Memory** window toolbar.
Right-click the main memory display.
From the shortcut menu, select **Save As**, then choose the format from the submenu.

To export the current state of memory to a binary editor:

Select the start address and number of bytes to save by editing the **Start Address** and **Size** fields in the **Memory** window toolbar.
Right-click the main memory display.
Choose **Export to Binary Editor** from the shortcut menu.

Note that subsequent modifications in the binary editor will not modify memory in the target.

Breakpoints window

The **Breakpoints** window manages the list of currently set breakpoints on the solution. Using the **Breakpoints** window, you can:

- Enable, disable, and delete existing breakpoints.
- Add new breakpoints.
- Show the status of existing breakpoints.

Breakpoints are stored in the session file, so they will be remembered each time you work on a particular project. When running in the debugger, you can set breakpoints on assembly code addresses. These low-level breakpoints appear in the **Breakpoints** window for the duration of the debug run but are not saved when you stop debugging.

When a breakpoint is reached, the matching breakpoint is highlighted in the **Breakpoints** window.

Breakpoints window layout

The **Breakpoints** window has a toolbar and a main breakpoint display.

Button	Description
	Create a new breakpoint using the **New Breakpoint** dialog.
	Toggle the selected breakpoint between enabled and disabled states.
	Remove the selected breakpoint.
	Move the insertion point to the statement where the selected breakpoint is set.
	Delete all breakpoints.
	Disable all breakpoints.
	Enable all breakpoints.
	Create a new breakpoint group and makes it active.

The main part of the **Breakpoints** window shows what breakpoints are set and the state they are in. You can organize breakpoints into folders, called *breakpoint groups*.

SEGGER Embedded Studio displays these icons to the left of each breakpoint:

Icon	Description

	Enabled breakpoint An enabled breakpoint will stop your program running when the breakpoint condition is met.
	Disabled breakpoint A disabled breakpoint will not stop the program when execution passes through it.
	Invalid breakpoint An invalid breakpoint is one where the breakpoint cannot be set; for example, no executable code is associated with the source code line where the breakpoint is set or the processor does not have enough hardware breakpoints.

Showing the Breakpoints window

To activate the Breakpoints window:

Choose **Breakpoints** > **Breakpoints** or press **Ctrl+Alt+B**.

Managing single breakpoints

You can manage breakpoints in the **Breakpoint** window.

To delete a breakpoint:

In the **Breakpoints** window, click the breakpoint to delete.

From the **Breakpoints** window toolbar, click the **Delete** Breakpoint} button.

To edit a breakpoint:

In the **Breakpoints** window, right-click the breakpoint to edit.

Choose **Edit Breakpoint** from the shortcut menu.

Edit the breakpoint in the **New Breakpoint** dialog.

To toggle the enabled state of a breakpoint:

In the **Breakpoints** window, right-click the breakpoint to enable or disable.

Choose **Enable/Disable Breakpoint** from the shortcut menu.

or

In the **Breakpoints** window, click the breakpoint to enable or disable.

Press **Ctrl+F9**.

Breakpoint groups

Breakpoints are divided into *breakpoint groups*. You can use breakpoint groups to specify sets of breakpoints that are applicable to a particular project in the solution or for a particular debug scenario. Initially, there is a single breakpoint group, named *Default*, to which all new breakpoints are added.

To create a new breakpoint group:

From the **Breakpoints** window toolbar, click the **New Breakpoint Group** button.

or

From the **Debug** menu, choose **Breakpoints** then **New Breakpoint Group**.

or

Right-click anywhere in the **Breakpoints** window.
Choose **New Breakpoint Group** from the shortcut menu.

In the **New Breakpoint Group** dialog, enter the name of the breakpoint group.

When you create a breakpoint, it is added to the active breakpoint group.

To make a group the active group:

In the **Breakpoints** window, right-click the breakpoint group to make active.
Choose **Set as Active Group** from the shortcut menu.

To delete a breakpoint group:

In the **Breakpoints** window, right-click the breakpoint group to delete.
Choose **Delete Breakpoint Group** from the shortcut menu.

You can enable all breakpoints within a group at once.

To enable all breakpoints in a group:

In the **Breakpoints** window, right-click the breakpoint group to enable.
Choose **Enable Breakpoint Group** from the shortcut menu.

You can disable all breakpoints within a group at once.

To disable all breakpoints in a group:

In the **Breakpoints** window, right-click the breakpoint group to disable.
Choose **Disable Breakpoint Group** from the shortcut menu.

Managing all breakpoints

You can delete, enable, or disable all breakpoints at once.

To delete all breakpoints:

Choose **Breakpoints** > **Clear All Breakpoints** or press **Ctrl+Shift+F9**.

or

On the **Breakpoints** window toolbar, click the **Delete All Breakpoints** button.

To enable all breakpoints:

Choose **Breakpoints** > **Enable All Breakpoints** or press **Ctrl+B, N**.

or

On the **Breakpoints** window toolbar, click the **Enable All Breakpoints** button.

To disable all breakpoints:

Choose **Breakpoints** > **Disable All Breakpoints** or press **Ctrl+B, X**.

or

On the **Breakpoints** window toolbar, click the **Disable All Breakpoints** button.

Call Stack window

The **Call Stack** window displays the list of function calls (stack frames) that were active when program execution halted. When execution halts, SEGGER Embedded Studio populates the call-stack window from the active (currently executing) task. For simple, single-threaded applications not using the SEGGER Embedded Studio tasking library, there is only a single task; but for multi-tasking programs that use the SEGGER Embedded Studio Tasking Library, there may be any number of tasks. SEGGER Embedded Studio updates the **Call Stack** window when you change the active task in the **Threads** window.

The **Call Stack** window has a toolbar and a main call-stack display.

Button	Description
	Move the insertion point to where the call was made to the selected frame.
	Set the debugger context to the selected stack frame.
	Move the debugger context down one stack to the called function.
	Move the debugger context up one stack to the calling function.
	Select the fields to display for each entry in the call stack.
	Set the debugger context to the most recent stack frame and move the insertion point to the currently executing statement.

The main part of the **Call Stack** window displays each unfinished function call (active stack frame) at the point when program execution halted. The most recent stack frame is displayed at the bottom of the list and the oldest is displayed at the top of the list.

SEGGER Embedded Studio displays these icons to the left of each function name:

Icon	Description
	Indicates the stack frame of the current task.
	Indicates the stack frame selected for the debugger context.
	Indicates that a breakpoint is active and when the function returns to its caller.

These icons can be overlaid to show, for instance, the debugger context and a breakpoint on the same stack frame.

Showing the call-stack window

To activate the Call Stack window:

Choose **View** > **Call Stack** or press **Ctrl+Alt+S**.

Configuring the call-stack window

Each entry in the **Call Stack** window displays the function name and, additionally, parameter names, types, and values. You can configure the **Call Stack** window to show varying amounts of information for each stack frame. By default, SEGGER Embedded Studio displays all information.

To show or hide a field:

1. On the **Call Stack** toolbar, click the **Options** button on the far right.
2. Select the fields to show, and deselect the ones that should be hidden.

Changing the debugger context

You can select the stack frame for the debugger context from the **Call Stack** window.

To move the debugger context to a specific stack frame:

In the **Call Stack** window, double-click the stack frame to move to.

or

In the **Call Stack** window, select the stack frame to move to.
On the **Call Stack** window's toolbar, click the **Switch To Frame** button.

or

In the **Call Stack** window, right-click the stack frame to move to.
Choose **Switch To Frame** from the shortcut menu.

The debugger moves the insertion point to the statement where the call was made. If there is no debug information for the statement at the call location, SEGGER Embedded Studio opens a disassembly window at the instruction.

To move the debugger context up one stack frame:

On the **Call Stack** window's toolbar, click the **Up One Stack Frame** button.

or

On the **Debug Location** toolbar, click the **Up One Stack Frame** button.

or

Press **Alt+-**.

The debugger moves the insertion point to the statement where the call was made. If there is no debug information for the statement at the call location, SEGGER Embedded Studio opens a disassembly window at the instruction.

To move the debugger context down one stack frame:

On the **Call Stack** window's toolbar, click the **Down One Stack Frame** button.

or

On the **Debug Location** toolbar, click the **Down One Stack Frame** button.

or

Press **Alt++**.

The debugger moves the insertion point to the statement where the call was made. If there is no debug information for the statement at the call location, SEGGER Embedded Studio opens a disassembly window at the instruction.

Setting a breakpoint on a return to a function

To set a breakpoint on return to a function:

In the **Call Stack** window, click the stack frame on the function to stop at on return.
On the **Build** toolbar, click the **Toggle Breakpoint** button.

or

In the **Call Stack** window, click the stack frame on the function to stop at on return.
Press **F9**.

or

In the **Call Stack** window, right-click the function to stop at on return.
Choose **Toggle Breakpoint** from the shortcut menu.

Threads window

The **Threads** window displays the set of executing contexts on the target processor structured as a set of queues.

To activate the Threads window:

Choose **View** > **More Debug Windows** > **Threads** or press **Ctrl+Alt+H**.

The window is populated using the threads script, which is a JavaScript program store in a file whose file-type project option is "Threads Script" (or is called `threads.js`) and is in the project that is being debugged.

When debugging starts, the threads script is loaded and the **function init()** is called to determine which columns are displayed in the **Threads** window.

When the application stops on a breakpoint, the function **update()** is called to create entries in the **Threads** window corresponding to the columns that have been created together with the saved execution context (register state) of the thread. By double-clicking one of the entries, the debugger displays its saved execution contextto put the debugger back into the default execution context, use **Show Next Statement**.

Writing the threads script

The threads script controls the **Threads** window with the **Threads** object.

The methods **Threads.setColumns** and **Threads.setSortByNumber** can be called from the **function init()**.

```
function init()
{
  Threads.setColumns("Name", "Priority", "State", "Time");
  Threads.setSortByNumber("Time");
}
```

The above example creates the named columns **Name**, **Priority**, **State**, and **Time** in the **Threads** window, with the **Time** column sorted numerically rather than alphabetically.

If you don't supply the **function init()** in the threads script, the **Threads** window will create the default columns **Name**, **Priority**, and **State**.

The methods **Threads.clear()**, **Threads.newqueue()**, and **Threads.add()** can be called from the **function update()**.

The **Threads.clear()** method clears the **Threads** window.

The **Threads.newqueue()** function takes a string argument and creates a new, top-level entry in the **Threads** window. Subsequent entries added to this window will go under this entry. If you don't call this, new entries will all be at the top level of the **Threads** window.

The **Threads.add()** function takes a variable number of string arguments, which should correspond to the number of columns displayed by the **Threads** window. The last argument to the **Threads.add()** function should be an array (possibly empty) containing the registers of the thread or, alternatively, a handle that can be supplied a call to the threads script **function getregs(handle)**, which will return an array when the thread is selected in the **Threads** window. The array containing the registers should have elements in the same order in which they are displayed in the CPU **Registers** displaytypically this will be in register-number order, e.g., **r0**, **r1**, and so on.

```
function update()
{
  Threads.clear();
  Threads.newqueue("My Tasks");
  Threads.add("Task1", "0", "Executing", "1000", [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]);
  Threads.add("Task2", "1", "Waiting", "2000", [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]);
}
```

The above example will create a fixed output on the **Threads** window and is here to demonstrate how to call the methods.

To get real thread state, you need to access the debugger from the threads script. To do this, you can use the JavaScript method **Debug.evaluate("expression")**, which will evaluate the string argument as a debug expression and return the result. The returned result will be an object if you evaluate an expression that denotes a structure or an array. If the expression denotes a structure, each field can be accessed by using its field name.

So, if you have structs in the application as follows

```
struct task {
  char *name;
  unsigned char priority;
  char *state;
  unsigned time;
  struct task *next;
  unsigned registers[17];
  unsigned thread_local_storage[4];
};

struct task task2 =
{
  "Task2",
   1,
   "Waiting",
   2000,
   0,
   { 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 },
   { 0,1,2,3 }
};

struct task task1 =
{
  "Task1",
  0,
  "Executing",
  1000,
  &task2,
  { 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 },
```

```
  { 0,1,2,3 }
};
```

you can **update()** the **Threads** window using the following:

```
task1 = Debug.evaluate("task1");
Threads.add(task1.name, task1.priority, task1.state, task1.time, task1.registers);
```

You can use pointers and C-style cast to enable linked-list traversal.

```
var next = Debug.evaluate("&task1");
while (next)
  {
    var xt = Debug.evaluate("*(struct task*)"+next);
    Threads.add(xt.name, xt.priority, xt.state, xt.time, xt.registers);
    next = xt.next;
  }
```

Note that, if the threads script goes into an endless loop, the debuggerand consequently SEGGER Embedded Studiowill become unresponsive and you will need to kill SEGGER Embedded Studio using a task manager. Therefore, the above loop is better coded as follows:

```
var next = Debug.evaluate("&task1");
var count = 0;
while (next && count < 10)
  {
    var xt = Debug.evaluate("*(struct task*)"+next);
    Threads.add(xt.name, xt.priority, xt.state, xt.time, xt.registers);
    next = xt.next;
    count++;
  }
```

You can speed up the **Threads** window update by not supplying the registers of the thread to the **Threads.add()** function. To do this, you should supply a handle/pointer to the thread as the last argument to the **Threads.add()** function. For example:

```
var next = Debug.evaluate("&task1");
var count = 0;
while (next && count < 10)
  {
    var xt = Debug.evaluate("*(struct task*)"+next);
    Threads.add(xt.name, xt.priority, xt.state, xt.time, next);
    next=xt.next;
    count++;
  }
```

When the thread is selected, the **Threads** window will call **getregs(x)** in the threads script. That function should return the array of registers, for example:

```
function getregs(x)
{
  return Debug.evaluate("((struct task*)"+x+")->registers");
}
```

If you use thread local storage, implementing the **gettls(x)** function enables you to return an expression for the debugger to evaluate when the base address of the thread local storage is accessed, for example:

```
function gettls(x)
{
  return "((struct task*)"+x+")->thread_local_storage";
}
```

The debugger may require the name of a thread which you can provide by implementing the **getname(x)** function, for example:

```
function getname(x)
{
  return Debug.evaluate("((struct task*)"+x+")->name");
}
```

Execution Profile window

The **Execution Profile** window shows a list of source locations and the number of times those source locations have been executed. This window is only available for targets that support the collection of jump trace information.

To activate the Execution Profile window:

Choose **View** > **More Debug Windows** > **Execution Profile** or press **Ctrl+T, P**.

The count value displayed is the number of times the first instruction of the source code location has been executed. The source locations displayed are target dependent: they could represent each statement of the program or each jump target of the program. If however the debugger is in intermixed or disassembly mode then the count values will be displayed on a per instruction basis.

The execution counts window is updated each time your program stops and the window is visible so if you have this window displayed then single stepping may be slower than usual.

Execution Trace window

The trace window displays historical information on the instructions executed by the target.

To activate the Trace window:

Choose **View** > **More Debug Windows** > **Execution Trace** or press **Ctrl+T, T**.

The type and number of the trace entries depends upon the target that is connected when gathering trace information. Some targets may trace all instructions, others may trace jump instructions, and some may trace modifications to variables. You'll find the trace capabilities of your target on the shortcut menu.

Each entry in the trace window has a unique number, and the lower the number the earlier the trace. You can click on the header to show earliest to latest or the latest to earliest trace entries. If a trace entry can have source code located to it then double-clicking the trace entry will show the appropriate source display.

Some targets may provide timing information which will be displayed in the ticks column.

The trace window is updated each time the debugger stops when it is visible so single stepping is likely to be slower if you have this window displayed.

Debug file search editor

When a program is built with debugging enabled, the debugging information contains the paths and filenames of all the source files for the program in order to allow the debugger to find them. If a program or library linked into the program is on a different machine than the one on which it was compiled, or if the source files were moved after the program was compiled, the debugger will not be able to find the source files.

In this situation, the simplest way to help SEGGER Embedded Studio find the source files is to add the directory containing the source files to one of its source-file search paths. Alternatively, if SEGGER Embedded Studio cannot find a source file, it will prompt you for its location and will record its new location in the source-file map.

Debug source-file search paths

Debug's source-file search paths can be used to help the debugger locate source files that are no longer located where they were at compile time. When a source file cannot be found, the search-path directories will be checked, in turn, to see if they contain the source file. SEGGER Embedded Studio maintains two debug source-file search paths:

Project-session search path:This path is for the current project session and does not apply to all projects.
The global search path:This system-wide path applies to all projects.

The project-session search path is checked before the global search path.

To edit the debug search paths:

Choose **Debug** > **Options** > **Search Paths**.

Debug source file map

If a source file cannot be found while debugging and the debugger has to prompt the user for its location, the results are stored in the debug source file map. The debug source file map simply correlates, or *maps*, the original pathnames to the new locations. When a file cannot be found at its original location or in the debug search paths, the debug source file map is checked to see if a new location has been recorded for the file or if the user has specified that the file does not exist. Each project session maintains its own source file map, the map is not shared by all projects.

To view the debug source file map:

Choose **Debug** > **Options** > **Search Paths**.

To remove individual entries from the debug source file map:

Choose **Debug** > **Options** > **Search Paths**.

Right-click the mapping to delete.

Choose **Delete Mapping** from the shortcut menu.

To remove all entries from the debug source file map:

Choose **Debug** > **Options** > **Search Paths**.

Right-click any mapping.

Choose **Delete All Mappings** from the shortcut menu.

Breakpoint expressions

The debugger can set breakpoints by evaluating simple C-like expressions. Note that the exact capabilities offered by the hardware to assist in data breakpointing will vary from target to target; please refer to the particular target interface you are using and the capabilities of your target silicon for exact details. The simplest expression supported is a symbol name. If the symbol name is a function, a breakpoint occurs when the first instruction of the symbol is about to be executed. If the symbol name is a variable, a breakpoint occurs when the symbol has been accessed; this is termed a *data breakpoint*. For example, the expression `x` will breakpoint when *x* is accessed. You can use a debug expression (see **Debug expressions**) as a breakpoint expression. For example, `x[4]` will breakpoint when element 4 of array *x* is accessed, and `@sp` will breakpoint when the *sp* register is accessed.

Data breakpoints can be specified, using the == operator, to occur when a symbol is accessed with a specific value. The expression `x == 4` will breakpoint when *x* is accessed and its value is 4. The operators <, >=, >;, >=, ==, and != can be used similarly. For example, `@sp <= 0x1000` will breakpoint when register **sp** is accessed and its value is less than or equal to 0x1000.

You can use the operator & to mask the value you wish to break on. For example, `(x & 1) == 1` will breakpoint when *x* is accessed and has an odd value.

You can use the operator && to combine comparisons. For example

```
(x >= 2) && (x <= 14)
```

will breakpoint when *x* is accessed and its value is between 2 and 14.

You can specify an arbitrary memory range using an array cast expression. For example, `(char[256])(0x1000)` will breakpoint when the memory region 0x10000x10FF is accessed.

You can specify an inverse memory range using the **!** operator. For example `!(char[256])(0x1000)` will breakpoint when memory outside the range 0x10000x10FF is accessed.

Debug expressions

The debugger can evaluate simple expressions that can be displayed in the **Watch** window or as a tool-tip in the code editor.

The simplest expression is an identifier the debugger tries to interpret in the following order:

- an identifier that exists in the scope of the current context.
- the name of a global identifier in the program of the current context.

Numbers can be used in expressions. Hexadecimal numbers must be prefixed with `0x`.

Registers can be referenced by prefixing the register name with `@`.

The standard C and C++ operators `!`, ~, `*`, `/`, %, +, `-`, >>, <<, <, <=, >, >=, ==, `|`, &, ^, &&, and `||` are supported on numeric types.

The standard assignment operators =, +=, `-=`, `*=`, `/=`, %=, >>, >>=, <<=, &=, `|=`, ^= are supported on numeric types.

The array subscript operator [] is supported on array and pointer types.

The structure access operator . is supported on structured types (this also works on pointers to structures), and -> works similarly.

The dereference operator (prefix *) is supported on pointers, the address-of (prefix &) and **sizeof** operators are supported.

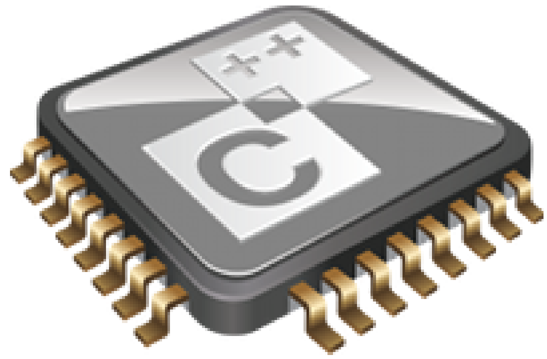
The `addressof(`*filename*`,` *linenumber*`)` operator will return the address of the specified source code line number.

Function calling with parameters and return results.

Casting to basic pointer types is supported. For example, (unsigned char *)0x300 can be used to display the memory at a given location.

Casting to basic array types is supported. For example, (unsigned char[256])0x100 can be used to reference a memory region.

Operators have the precedence and associativity one expects of a C-like programming language.



Command-line options

This section describes the command-line options accepted by SEGGER Embedded Studio.

Usage

emStudio [*options*] [*files*]

-D (Define macro)

Syntax

-D *macro=value*

Description

Define a SEGGER Embedded Studio macro value.

-noclang (Disable Clang support)

Syntax

-noclang

Description

Disable Clang support.

-packagesdir (Specify packages directory)

Syntax

-packagesdir *dir*

Description

Override the default value of the **$(PackagesDir)** macro.

-permit-multiple-studio-instances (Permit multiple studio instances)

Syntax

-permit-multiple-studio-instances

Description

Allow multiple instances of SEGGER Embedded Studio to run at the same time. This behaviour can also be enabled using the **Environment** > **Startup Options** > **Allow Multiple SEGGER Embedded Studios** environment option.

-rootuserdir (Set the root user data directory)

Syntax

-rootuserdir *dir*

Description

Set the SEGGER Embedded Studio root user data directory.

-save-settings-off (Disable saving of environment settings)

Syntax

-save-settings-off

Description

Disable the saving of modified environment settings.

-set-setting (Set environment setting)

Syntax

-set-setting *environment_setting=value*

Description

Sets an environment setting to a specified value. For example:

```
-set-setting "Environment/Build/Show Command Lines=Yes"
```

-templatesfile (Set project templates path)

Syntax

-templatesfile *path*

Description

Sets the search path for finding project template files.

Uninstalling SEGGER Embedded Studio

This section describes how to completely uninstall SEGGER Embedded Studio for each supported operating system:

- **Uninstalling SEGGER Embedded Studio from Windows**
- **Uninstalling SEGGER Embedded Studio from Mac OS X**
- **Uninstalling SEGGER Embedded Studio from Linux**

Uninstalling SEGGER Embedded Studio from Windows

Removing user data and settings

The uninstaller does not remove any user data such as settings or installed packages. To completely remove the user data you will need to carry out the following operations for each user that has used SEGGER Embedded Studio on your system.

To remove user data using SEGGER Embedded Studio:

1. Start SEGGER Embedded Studio.
2. Click **Tools** > **Admin** > **Remove All User Data...**

Alternatively, if SEGGER Embedded Studio has already been uninstalled you can manually remove the user data as follows:

1. Click the Windows Start button.

2. Type *%LOCALAPPDATA%* in the search field and press enter to open the local application data folder.
3. Open the *SEGGER* folder.
4. Open the *SEGGER Embedded Studio* folder.
5. Delete the *v3* folder.
6. If you want to delete user data for all versions of the software, delete the *SEGGER Embedded Studio* folder as well.

Uninstalling SEGGER Embedded Studio

To uninstall SEGGER Embedded Studio:

1. If SEGGER Embedded Studio is running, click **File** > **Exit** to shut it down.
2. Click the Start Menu and select Control Panel. The Control Panel window will open.
3. In the Control Panel window, click the **Uninstall a program** link under the Programs section.
4. From the list of currently installed programs, select **SEGGER Embedded Studio 3.10i**.
5. To begin the uninstall, click the **Uninstall** button at the top of the list.

Uninstalling SEGGER Embedded Studio from Mac OS X

Removing user data and settings

Uninstalling does not remove any user data such as settings or installed packages. To completely remove the user data you will need to carry out the following operations for each user that has used SEGGER Embedded Studio on your system.

To remove user data using SEGGER Embedded Studio:

1. Start SEGGER Embedded Studio.
2. Click **Tools** > **Admin** > **Remove All User Data...**

Alternatively, if SEGGER Embedded Studio has already been uninstalled you can manually remove the user data as follows:

1. Open Finder.
2. Go to the *$HOME/Library/SEGGER/SEGGER Embedded Studio* directory.
3. Drag the *v3* folder to the Trash.
4. If you want to delete user data for all versions of the software, drag the *SEGGER Embedded Studio* folder to the Trash as well.

Uninstalling SEGGER Embedded Studio

To uninstall SEGGER Embedded Studio:

1. If SEGGER Embedded Studio is running, shut it down.
2. Open the *Applications* folder in Finder.
3. Drag the *SEGGER Embedded Studio 3.10i* folder to the Trash.

Uninstalling SEGGER Embedded Studio from Linux

Removing user data and settings

The uninstaller does not remove any user data such as settings or installed packages. To completely remove the user data you will need to carry out the following operations for each user that has used SEGGER Embedded Studio on your system.

To remove user data using SEGGER Embedded Studio:

1. Start SEGGER Embedded Studio.
2. Click **Tools** > **Admin** > **Remove All User Data...**

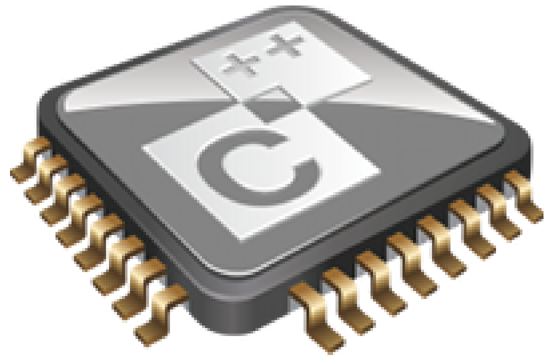
Alternatively, if SEGGER Embedded Studio has already been uninstalled you can manually remove the user data as follows:

1. Open a terminal window or file browser.
2. Go to the *$HOME/.segger/SEGGER Embedded Studio* directory.
3. Delete the *v3* directory.
4. If you want to delete user data for all versions of the software, delete the *SEGGER Embedded Studio* directory as well.

Uninstalling SEGGER Embedded Studio

To uninstall SEGGER Embedded Studio:

1. If SEGGER Embedded Studio is running, click **File** > **Exit** to shut it down.
2. Open a terminal window.
3. Go to the SEGGER Embedded Studio bin directory (this is */usr/share/segger_embedded_studio_3.10i/bin* by default).
4. Run *sudo ./uninstall* to start the uninstaller.



ARM target support

When a target-specific executable project is created using the **New Project Wizard**, the following default files are added to the project:

*Target*_`Startup.s` The target-specific startup code. See **Target startup code**.
`crt0.s` The SEGGER Embedded Studio standard C runtime. See **Startup code**.
*Target*_MemoryMap.xml The target-specific memory map file for the board. See **Section Placement**. Note that, for some targets, a general linker placement file may not be suitable. In these cases, there will be two memory-map files: one for a flash build and one for a RAM build.
`flash_placement.xml` The linker placement file for a flash build.
`sram_placement.xml` The linker placement file for a RAM build.

Initially, shared versions of these files are added to the project. If you want to modify any these shared files, select the file in the **Project Explorer** and then click the **Import** option from the shortcut menu. This will copy a writable version of the file into your project directory and change the path in the **Project Explorer** to that of the local version. You can then make changes to the local file without affecting the shared copy of it.

The following list describes the typical flow of a C program created with SEGGER Embedded Studio's project templates:

The processor starts executing at address 0x0000000, which is the reset exception vector. The exception-vector table can be found in the target-specific startup code (see **Target startup code**), and is put into the program section **.vectors**, which is positioned at address 0x00000000 by the target-specific memory-map file.
The processor jumps to the **reset_handler** label in the target-specific startup code, which configures the target (see **Target startup code**).

When the target is configured, the target-specific startup code jumps to the **_start** entry point in the C runtime code, which sets up the C runtime environment (see **Startup code**).

When the C runtime environment has been set up, the C runtime code jumps to the C entry-point function, **main**.

When the program returns from main, it re-enters the C runtime code, executes the destructors and enters an endless loop.

Target startup code

The following section describes the role of the target-specific startup code.

When you create a new project to produce an executable file using a target-specific project template, a file containing the default startup code for the target will be added to the project. Initially, a shared version of this file will be added to the project; if you want to modify this file, select the file in the **Project Explorer** and select **Import** to copy the file to your project directory.

ARM Target startup code

The target startup file typically consists of the exception vector table and the default set of exception handlers.

_vectors This is the exception vector table. It is put into its own **.vectors** section in order to ensure that it is always placed at address 0x00000000. The vector table contains jump instructions to the particular exception handlers. It is recommended that absolute jump instructions are used `ldr pc, [pc, #handler_address - . - 8]` rather than relative branch instructions `b handler_address` since many devices shadow the memory at address zero to start execution but the program will be linked to run at a different address.

reset_handler This is the main reset handler function and typically is the main entry point of an executable. The reset handler will usually carry out any target-specific initialization and then will jump to the **_start** entry point. In a C system, the **_start** entry point is in the **crt0.s** file. During development it is usual to replace this jump with an endless loop which will stop the device running potentially dangerous in-development code directly out of reset.

undef_handler This is the default, undefined-instruction exception handler.*

swi_handler This is the default, software-interrupt exception handler.*

pabort_handler This is the default, prefetch-abort exception handler.*

dabort_handler This is the default, data-abort exception handler.*

irq_handler This is the default, IRQ-exception handler.*

fiq_handler This is the default, FIQ-exception handler.*

* Declared as a weak symbol to allow the user to override the implementation.

Note that ARM exception handlers must be written in ARM assembly code. The CPU or board support package of the project you have created will typically supply an ARM assembly-coded **irq_handler** implementation that will enable you to write interrupt service routines as C functions.

Cortex-M Target startup code

The target startup file typically consists of the exception vector table and the default set of exception handlers.

_vectors This is the exception vector table. It is put into its own **.vectors** section in order to ensure that it is always placed at address 0x00000000.

The vector table is structured as follows:

The first entry is the initial value of the stack pointer.
The second entry is the address of the reset handler function. The reset handler will usually carry out any target-specific initialization and then jump to the **_start** entry point. In a C system, the **_start** entry point is in the `crt0.s` file. During development it is usual to replace this jump with an endless loop which will stop the device running potentially dangerous in-development code directly out of reset.
The following 15 entries are the addresses of the standard Cortex-M exception handlers ending with the **SysTick_ISR** entry.
Subsequent entries are addresses of device-specific interrupt sources and their associated handlers.

For each exception handler, a weak symbol is declared that will implement an endless loop. You can implement your own exception handler as a regular C function. Note that the name of the C function must match the name in the startup code e.g. **void SysTick_ISR(void)**. You can use the C preprocessor to rename the symbol in the startup code if you have existing code with different exception handler names e.g. **SysTick_ISR=SysTick_Handler**.

Startup code

The following section describes the role of the C runtime-startup code, **crt0.s** (and the Cortex-M3/Thumb-2 equivalent **thumb_crt0.s**).

When you create a new project to produce an executable file using a target-specific project template, the **crt0.s** file is added to the project. Initially, a shared version of this file is added to the project. If you want to modify this file, right-click it in the **Project Explorer** and then select **Import** from the shortcut menu to copy the file to your project directory.

The entry point of the C runtime-startup code is **_start**. In a typical system, this will be called by the target-specific startup code after it has initialized the target.

The C runtime carries out the following actions:

- Initialize the stacks.
- If required, copy the contents of the **.data** (initialized data) section from non-volatile memory.
- If required, copy the contents of the **.fast** section from non-volatile memory to SRAM.
- Initialize the **.bss** section to zero.
- Initialize the heap.
- Call constructors.
- If compiled with **FULL_LIBRARY**, get the command line from the host using **debug_getargs** and set registers to supply **argc** and **argv** to **main**.
- Call the **main** entry point.

On return from **main** or when **exit** is called

- If compiled with **FULL_LIBRARY**, call destructors.
- If compiled with **FULL_LIBRARY**, call **atexit** functions.
- If compiled with **FULL_LIBRARY**, call **debug_exit** while supplying the return result from **main**.
- Wait in exit loop.

Program sections

The following program sections are used for the C runtime in section-placement files:

Section name	Description
.vectors	The exception vector table.
.init	Startup code that runs before the call to the application's **main** function.
.ctors	Static constructor function table.
.dtors	Static destructor function table.
.text	The program code.
.fast	Code to copy from flash to RAM for fast execution.

.data	The initialized static data.
.bss	The zeroed static data.
.rodata	The read-only constants and literals of the program.
.ARM.exidx	The C++ exception table.

Stacks

The ARM maintains six separate stacks. The position and size of these stacks are specified in the project's section-placement or memory-map file by the following program sections:

Section name	Linker size symbol	Description
`.stack`	`__STACKSIZE__`	System and User mode stack.
`.stack_svc`	`__STACKSIZE_SVC__`	Supervisor mode stack
`.stack_irq`	`__STACKSIZE_IRQ__`	IRQ mode stack
`.stack_fiq`	`__STACKSIZE_FIQ__`	FIQ mode stack
`.stack_abt`	`__STACKSIZE_ABT__`	Abort mode stack
`.stack_und`	`__STACKSIZE_UND__`	Undefined mode stack

For Cortex-M devices the following stacks and linker symbol stack sizes are defined:

Section name	Linker size symbol	Description
`.stack`	`__STACKSIZE__`	Main stack.
`.stack_process`	`__STACKSIZE_PROCESS__`	Process stack.

The **crt0.s** startup code references these sections and initializes each of the stack-pointer registers to point to the appropriate location. To change the location in memory of a particular stack, the section should be moved to the required position in the section-placement or memory-map file.

Should your application not require one or more of these stacks, you can remove those sections from the memory-map file or set the size to 0 and remove the initialization code from the `crt0.s` file.

The .data section

The **.data** section contains the initialized data. If the run address is different from the load address, as it would be in a flash-based application in order to allow the program to run from reset, the `crt0.s` startup code will copy the **.data** section from the load address to the run address before calling the **main** entry point.

The .fast section

For performance reasons, it is a common requirement for embedded systems to run critical code from fast memory; the **.fast** section can be used to simplify this. If the **.fast** section's run address is different from the load address, the `crt0.s` startup code will copy the **.fast** section from the load address to the run address before calling the **main** entry point.

The .bss Section

The **.bss** section contains the zero-initialized data. The startup code in `crt0.s` references the **.bss** section and sets its contents to zero.

The heap

The position and size of the heap is specified in the project's section-placement or memory-map file by the **.heap** program section.

The startup code in `crt0.s` references this section and initializes the heap. To change the position of the heap, the section should be moved to the required position in the section-placement or memory-map file.

There is a **Heap Size** linker project option you can modify in order to alter the heap size. For compatibility with earlier versions of SEGGER Embedded Studio, you can also specify the heap size using the heap section's **Size** attribute in the section-placement or memory-map file.

Should your application not require the heap functions, you can remove the heap section from the memory-map file or set the size to zero and remove the heap-initialization code from the `crt0.s` file.

Section Placement

SEGGER Embedded Studio's memory-map files are XML files and are used

Linking:by the linker, to describe how to lay out a program in memory.
Loading:by the loader, to check whether a program being downloaded will actually fit into the target's memory.
Debugging:by the debugger, to describe the location and types of memory a target has. This information is used to decide how to debug the programfor example, whether to set hardware or software breakpoints on particular memory location.

Section placement files map program sections used in your program into the memory spaces defined in the memory map. For instance, it's common for code and read-only data to be programmed into non-volatile flash memory, whereas read-write data needs to be mapped onto either internal or external RAM.

Memory map files are provided in the CPU support package you are using and are referenced in executable projects by the **Memory Map File** project option. Section-placement files are provided in the base SEGGER Embedded Studio distribution.

ARM section placement

The following placement files are supplied for ARM targets:

File	Description
`flash_placement.xml`	Single FLASH segment with internal RAM segment and optional external RAM segment.
`flash_run_text_from_ram_placement.xml`	Single FLASH segment with internal RAM segment and optional external RAM segments. Text section is copied from FLASH to RAM.
`internal_sram_placement.xml`	Single internal RAM segment.
`flash_placement.xml`	Two FLASH segments with internal RAM segment and optional external RAM segment.
`internal_sram_placement.xml`	Internal RAM segment and optional external RAM segment.

Cortex-M section placement

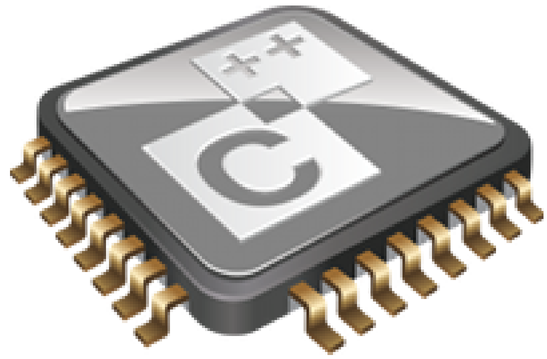
The following placement files are supplied for Cortex-M targets:

File	Description
`flash_placement.xml`	Two FLASH segments and two RAM segments.
`flash_placement2.xml`	One FLASH segment and two RAM segments.

`flash_to_ram_placement.xml`	One FLASH segment and one RAM segment. Text section is copied from FLASH to RAM.
`ram_placement.xml`	Two RAM segments.

The memory segments defined in the section placement files have macro-expandable names which can be defined using the **Section Placement Macros** project option.

Some of the section placement files have a macro-expandable start attribute in the first program section. You can use this to reserve space at the beginning of the memory segment.



C Library User Guide

This section describes the library and how to use and customize it.

The libraries supplied with SEGGER Embedded Studio have all the support necessary for input and output using the standard C functions **printf** and **scanf**, support for the **assert** function, both 32-bit and 64-bit floating point, and are capable of being used in a multi-threaded environment. However, to use these facilities effectively you will need to customize the low-level details of *how* to input and output characters, what to do when an assertion fails, how to provide protection in a multithreaded environment, and how to use the available hardware to the best of its ability.

Floating point

The SEGGER Embedded Studio C library uses IEEE floating point format as specified by the ISO 60559 standard with restrictions.

This library favors code size and execution speed above absolute precision. It is suitable for applications that need to run quickly and not consume precious resources in limited environments. The library does not implement features rarely used by simple applications: floating point exceptions, rounding modes, and subnormals.

NaNs and infinities are supported and correctly generated. The only rounding mode supported is round-to-nearest. Subnormals are always flushed to a correctly-signed zero. The mathematical functions use stable approximations and do their best to cater ill-conditioned inputs.

Single and double precision

SEGGER Embedded Studio C allows you to choose whether the **double** data type uses the IEC 60559 32-bit or 64-bit format. The following sections describe the details of why you would want to choose a 32-bit **double** rather than a 64-bit **double** in many circumstances.

Why choose 32-bit doubles?

Many users are surprised when using **float** variables exclusively that sometimes their calculations are compiled into code that calls for **double** arithmetic. They point out that the C standard allows **float** arithmetic to be carried out only using **float** operations and not to automatically promote to the **double** data type of classic K&R C.

This is valid point. However, upon examination, even the simplest calculations can lead to **double** arithmetic. Consider:

```
// Compute sin(2x)
float sin_two_x(float x)
{
  return sinf(2.0 * x);
}
```

This looks simple enough. We're using the **sinf** function which computes the sine of a **float** and returns a **float** result. There appears to be no mention of a **double** anywhere, yet the compiler generates code that calls **double** support routinesbut why?

The answer is that the constant **2.0** is a **double** constant, not a **float** constant. That is enough to force the compiler to convert both operands of the multiplication to **double** format, perform the multiplication in **double** precision, and then convert the result back to **float** precision. To avoid this surprise, the code should have been written:

```
// Compute sin(2x)
float sin_two_x(float x)
{
  return sinf(2.0F * x);
}
```

This uses a single precision floating-point constant **2.0F**. It's all too easy to forget to correctly type your floating-point constants, so if you compile your program with **double** meaning the same as **float**, you can forget all about adding the '**F**' suffix to your floating point constants.

As an aside, the C99 standard is very strict about the way that floating-point is implemented and the latitude the compiler has to rearrange and manipulate expressions that have floating-point operands. The compiler cannot second-guess user intention and use a number of useful mathematical identities and algebraic simplifications because in the world of IEC 60559 arithmetic many algebraic identities, such as **x * 1 = x**, do not hold when **x** takes one of the special values NaN, infinity, or negative zero.

More reasons to choose 32-bit doubles

Floating-point constants are not the only silent way that **double** creeps into your program. Consider this:

```
void write_results(float x)
{
  printf("After all that x=%f\\n", x);
}
```

Again, no mention of a **double** anywhere, but **double** support routines are now required. The reason is that ISO C requires that **float** arguments are promoted to **double** when they are passed to the non-fixed part of variadic functions such as **printf**. So, even though your application may never mention **double**, **double** arithmetic may be required simply because you use **printf** or one of its near relatives.

If, however, you compile your code with 32-bit doubles, then there is no requirement to promote a **float** to a **double** as they share the same internal format.

Why choose 64-bit doubles?

If your application requires very accurate floating-point, more precise than the seven decimal digits supported by the **float** format, then you have little option but to use **double** arithmetic as there is no simple way to increase the precision of the **float** format. The **double** format delivers approximately 15 decimal digits of precision.

Multithreading

The SEGGER Embedded Studio libraries support multithreading, for example, where you are using CTL or a third-party real-time operating system (RTOS).

Where you have single-threaded processes, there is a single flow of control. However, in multithreaded applications there may be several flows of control which access the same functions, or the same resources, concurrently. To protect the integrity of resources, any code you write for multithreaded applications must be *reentrant* and *thread-safe*.

Reentrancy and thread safety are both related to the way functions in a multithreaded application handle resources.

Reentrant functions

A reentrant function does not hold static data over successive calls and does not return a pointer to static data. For this type of function, the caller provides all the data that the function requires, such as pointers to any workspace. This means that multiple concurrent calls to the function do not interfere with each other, that the function can be called in mainline code, and that the function can be called from an interrupt service routine.

Thread-safe functions

A thread-safe function protects shared resources from concurrent access using locks. In C, local variables are held in processor registers or are on the stack. Any function that does not use static data, or other shared resources, is thread-safe. In general, thread-safe functions are safe to call from any thread but cannot be called directly, or indirectly, from an interrupt service routine.

Thread safety in the SEGGER Embedded Studio library

In the SEGGER Embedded Studio C library:

- some functions are inherently thread-safe, for example **strcmp**.
- some functions, such as **malloc**, are not thread-safe by default but can be made thread-safe by implementing appropriate lock functions.
- other functions are only thread-safe if passed appropriate arguments, for example **tmpnam**.
- some functions are never thread-safe, for example **setlocale**.

We define how the functions in the C library can be made thread-safe if needed. If you use a third-party library in a multi-threaded system and combine it with the SEGGER Embedded Studio C library, you will need to ensure that the third-party library can be made thread-safe in just the same way that the SEGGER Embedded Studio C library can be made thread-safe.

Implementing mutual exclusion in the C library

The SEGGER Embedded Studio C library ships as standard with callouts to functions that provide thread-safety in a multithreaded application. If your application has a single thread of execution, the default implementation of these functions does nothing and your application will run without modification.

If your application is intended for a multithreaded environment and you wish to use the SEGGER Embedded Studio C library, you must implement the following locking functions:

- **__heap_lock** and **__heap_unlock** to provide thread-safety for all heap operations such as **malloc**, **free**, and **realloc**.
- **__printf_lock** and **__printf_unlock** to provide thread-safety for **printf** and relatives.
- **__scanf_lock** and **__scanf_unlock** to provide thread-safety for **scanf** and relatives.
- **__debug_io_lock** and **__debug_io_unlock** to provide thread-safety for semi-hosting support in the SEGGER Embedded Studio I/O function.

If you use a third-party RTOS with the SEGGER Embedded Studio C library, you will need to use whatever your RTOS provides for mutual exclusion, typically a semaphore, a mutex, or an event set.

Input and output

The C library provides all the standard C functions for input and output except for the essential items of where to output characters printed to **stdout** and where to read characters from **stdin**.

If you want to output to a UART, to an LCD, or input from a keyboard using the standard library print and scan functions, you need to customize the low-level input and output functions.

Customizing putchar

To use the standard output functions **putchar**, **puts**, and **printf**, you need to customize the way that characters are written to the standard output device. These output functions rely on a function **__putchar** that outputs a character and returns an indication of whether it was successfully written.

The prototype for **__putchar** is

```
int __putchar(int ch);
```

Sending all output to the SEGGER Embedded Studio virtual terminal

You can send all output to the SEGGER Embedded Studio virtual terminal by supplying the following implementation of the**__putchar** function in your code:

```
#include <debugio.h>

int __putchar(int ch)
{
  return debug_putchar(ch);
}
```

This hands off output of the character **ch** to the low-level debug output routine, **debug_putchar**.

Whilst this is an adequate implementation of **__putchar**, it does consume stack space for an unnecessary nested call and associated register saving. A better way of achieving the same result is to define the low-level symbol for **__putchar** to be equivalent to the low-level symbol for **debug_putchar**. To do this, we need to instruct the linker to make the symbols equivalent.

Select the project node in the **Project Explorer**.
Display the **Properties Window**.
Enter the text `__putchar=debug_putchar` into the **Linker** > **Linker Symbol Definitions** property of the **Linker Options** group.

Sending all output to another device

If you need to output to a physical device, such as a UART, the following notes will help you:

If the character cannot be written for any reason, **putchar** *must* return **EOF**. Just because a character can't be written immediately is not a reason to return **EOF**: you can busy-wait or tasking (if applicable) to wait until the character is ready to be written.
The higher layers of the library do not translate C's end of line character '**\\n**' before passing it to **putchar**. If you are directing output to a serial line connected to a terminal, for instance, you will most likely need to output a carriage return and line feed when given the character '**\\n**' (ASCII code 10).

The standard functions that perform input and output are the **printf** and **scanf** functions.These functions convert between internal binary and external printable data. In some cases, though, you need to read and write formatted data on other channels, such as other RS232 ports. This section shows how you can extend the I/O library to best implement these function.

Classic custom printf-style output

Assume that we need to output formatted data to two UARTs, numbered 0 and 1, and we have a functions **uart0_putc** and **uart1_putc** that do just that and whose prototypes are:

```
int uart0_putc(int ch, __printf_t *ctx);
int uart1_putc(int ch, __printf_t *ctx);
```

These functions return a positive value if there is no error outputting the character and EOF if there was an error. The second parameter, `ctx`, is the *context* that the high-level formatting routines use to implement the C standard library functions.

Using a classic implementation, you would use **sprintf** to format the string for output and then output it:

```
void uart0_printf(const char *fmt, ...)
{
  char buf[80], *p;
  va_list ap;
  va_start(ap, fmt);
  vsnprintf(buf, sizeof(buf), fmt, ap);
  for (p = buf; *p; ++p)
    uart0_putc(*p, 0);  // null context
  va_end(ap);
}
```

We would, of course, need an identical routine for outputting to the other UART. This code is portable, but it requires an intermediate buffer of 80 characters. On small systems, this is quite an overhead, so we could reduce the buffer size to compensate. Of course, the trouble with that means that the maximum number of characters that can be output by a single call to **uart0_printf** is also reduced. What would be good is a way to output characters to one of the UARTs without requiring an intermediate buffer.

SEGGER Embedded Studio printf-style output

SEGGER Embedded Studio provides a solution for just this case by using some internal functions and data types in the SEGGER Embedded Studio library. These functions and types are define in the header file <__vfprintf.h>.

The first thing to introduce is the **__printf_t** type which captures the current state and parameters of the format conversion:

```
typedef struct __printf_tag
{
  size_t charcount;
  size_t maxchars;
  char *string;
  int (*output_fn)(int, struct __printf_tag *ctx);
```

```
} __printf_t;
```

This type is used by the library functions to direct what the formatting routines do with each character they need to output. If `string` is non-zero, the character is appended is appended to the string pointed to by **string**; if `output_fn` is non-zero, the character is output through the function **output_fn** with the context passed as the second parameter.

The member **charcount** counts the number of characters currently output, and **maxchars** defines the maximum number of characters output by the formatting routine **__vfprintf**.

We can use this type and function to rewrite **uart0_printf**:

```
int uart0_printf(const char *fmt, ...)
{
  int n;
  va_list ap;
  __printf_t iod;
  va_start(ap, fmt);
  iod.string = 0;
  iod.maxchars = INT_MAX;
  iod.output_fn = uart0_putc;
  n = __vfprintf(\&iod, fmt, ap);
  va_end(ap);
  return n;
}
```

This function has no intermediate buffer: when a character is ready to be output by the formatting routine, it calls the **output_fn** function in the descriptor **iod** to output it immediately. The maximum number of characters isn't limited as the **maxchars** member is set to **INT_MAX**. if you wanted to limit the number of characters output you can simply set the **maxchars** member to the appropriate value before calling **__vfprintf**.

We can adapt this function to take a UART number as a parameter:

```
int uart_printf(int uart, const char *fmt, ...)
{
  int n;
  va_list ap;
  __printf_t iod;
  va_start(ap, fmt);
  iod.is_string = 0;
  iod.maxchars = INT_MAX;
  iod.output_fn = uart ? uart1_putc : uart0_putc;
  n = __vfprintf(\&iod, fmt, ap);
  va_end(ap);
  return n;
}
```

Now we can use:

```
uart_printf(0, "This is uart %d\n...", 0);
uart_printf(1, "..and this is uart %d\n", 1);
```

__vfprintf returns the actual number of characters printed, which you may wish to dispense with and make the **uart_printf** routine return **void**.

Extending input functions

The formatted input functions would be implemented in the same manner as the output functions: read a string into an intermediate buffer and parse using **sscanf**. However, we can use the low-level routines in the SEGGER Embedded Studio library for formatted input without requiring the intermediate buffer.

The type **__stream_scanf_t** is:

```
typedef struct
{
  char is_string;
  int (*getc_fn)(void);
  int (*ungetc_fn)(int);
} __stream_scanf_t;
```

The function **getc_fn** reads a single character from the UART, and **ungetc_fn** pushes back a character to the UART. You can push at most one character back onto the stream.

Here's an implementation of functions to read and write from a single UART:

```
static int uart0_ungot = EOF;

int uart0_getc(void)
{
  if (uart0_ungot)
    {
      int c = uart0_ungot;
      uart0_ungot = EOF;
      return c;
    }
  else
    return read_char_from_uart(0);
}
```

```
int uart0_ungetc{int c)
{
  uart0_ungot = c;
}
```

You can use these two functions to perform formatted input using the UART:

```
int uart0_scanf(const char *fmt, ...)
{
  __stream_scanf_t iod;
  va_list a;
  int n;
  va_start(a, fmt);
  iod.is_string = 0;
  iod.getc_fn = uart0_getc;
  iod.ungetc_fn = uart0_ungetc;
  n = __vfscanf((__scanf_t *)\&iod, (const unsigned char *)fmt, a);
  va_end(a);
  return n;
}
```

Using this template, we can add functions to do additional formatted input from other UARTs or devices, just as we did for formatted output.

Locales

The SEGGER Embedded Studio C library supports wide characters, multi-byte characters and locales. However, as not all programs require full localization, you can tailor the exact support provided by the SEGGER Embedded Studio C library to suit your application. These sections describe how to add new locales to your application and customize the runtime footprint of the C library.

Unicode, ISO 10646, and wide characters

The ISO standard 10646 is identical to the published Unicode standard and the SEGGER Embedded Studio C library uses the Unicode 6.2 definition as a base. Hence, whenever you see the term Unicode in this document, it is equivalent to Unicode 6.2 and ISO/IEC 10646:2011.

The SEGGER Embedded Studio C library supports both 16-bit and 32-bit wide characters, depending upon the setting of wide character width in the project.

When compiling with 16-bit wide characters, all characters in the Basic Multilingual Plane are representable in a single `wchar_t` (values 0 through 0xFFFF). When compiling with 32-bit wide characters, all characters in the Basic Multilingual Plane and planes 1 through 16 are representable in a single **wchar_t** (values 0 through 0x10FFFF).

The wide character type will hold Unicode code points in a locale that is defined to use Unicode and character type functions such as **iswalpha** will work correctly on all Unicode code points.

Multi-byte characters

SEGGER Embedded Studio supports multi-byte encoding and decoding of characters. Most new software on the desktop uses Unicode internally and UTF-8 as the external, on-disk encoding for files and for transport over 8-bit mediums such as network connections.

However, in embedded software there is still a case to use code pages, such as ISO-Latin1, to reduce the footprint of an application whilst also providing extra characters that do not form part of the ASCII character set.

The SEGGER Embedded Studio C library can support both models and you can choose a combination of models, dependent upon locale, or construct a custom locale.

The standard C and POSIX locales

The standard C locale is called simply C. In order to provide POSIX compatibility, the name POSIX is a synonym for C.

The C locale is fixed and supports only the ASCII character set with character codes 0 through 127. There is no multi-byte character support, so the character encoding between wide and narrow characters is simply one-to-one: a narrow character is converted to a wide character by zero extension. Thus, ASCII encoding of narrow characters is compatible with the ISO 10646 (Unicode) encoding of wide characters in this locale.

Additional locales in source form

The SEGGER Embedded Studio C library provides only the C locale; if you need other locales, you must provide those by linking them into your application. We have constructed a number of locales from the Unicode Common Locale Data Repository (CLDR) and provided them in source form in the `$(StudioDir)/source` folder for you to include in your application.

A C library locale is divided into two parts:

- the locale's date, time, numeric, and monetary formatting information
- how to convert between multi-byte characters and wide characters by the functions in the C library.

The first, the locale data, is independent of how characters are represented. The second, the code set in use, defines how to map between narrow, multi-byte, and wide characters.

Installing a locale

If the locale you request using **setlocale** is neither C nor POSIX, the C library calls the function **__user_find_locale** to find a user-supplied locale. The standard implementation of this function is to return a null pointer which indicates that no additional locales are installed and, hence, no locale matches the request.

The prototype for **__user_find_locale** is:

```
const __RAL_locale_t *__user_find_locale(const char *locale);
```

The parameter **locale** is the locale to find; the locale name is terminated either by a zero character *or* by a semicolon. The locale name, up to the semicolon or zero, is identical to the name passed to **setlocale** when you select a locale.

Now let's install the Hungarian locale using both UTF-8 and ISO 8859-2 encodings. The UTF-8 codecs are included in the SEGGER Embedded Studio C library, but the Hungarian locale and the ISO 8859-2 codec are not.

You will find the file **locale_hu_HU.c** in the source directory as described in the previous section. Add this file to your project.

Although this adds the data needed for the locale, it does not make the locale available for the C library: we need to write some code for **__user_find_locale** to return the appropriate locales.

To create the locales, we need to add the following code and data to tie everything together:

```
#include <__crossworks.h>

static const __RAL_locale_t hu_HU_utf8 = {
  "hu_HU.utf8",
  &locale_hu_HU,
  &codeset_utf8
};

static const __RAL_locale_t hu_HU_iso_8859_2 = {
  "hu_HU.iso_8859_2",
  &locale_hu_HU,
  &codeset_iso_8859_2
};

const __RAL_locale_t *
__user_find_locale(const char *locale)
{
  if (__RAL_compare_locale_name(locale, hu_HU_utf8.name) == 0)
    return &hu_HU_utf8;
  else if (__RAL_compare_locale_name(locale, hu_HU_iso_8859_2.name) == 0)
    return &hu_HU_iso_8859_2;
  else
    return 0;
}
```

The function `__RAL_compare_locale_name` matches locale names up to a terminating null character, or a semicolon (which is required by the implementation of `setlocale` in the C library when setting multiple locales using LC_ALL).

In addition to this, you must provide a buffer, `__user_locale_name_buffer`, for locale names encoded by **setlocale**. The buffer must be large enough to contain five locale names, one for each category. In the above example, the longest locale name is `hu_HU.iso_8859_2` which is 16 characters in length. Using this information, buffer must be at least (16+1)5 = 85 characters in size:

```
const char __user_locale_name_buffer[85];
```

Setting a locale directly

Although we support **setlocale** in its full generality, most likely you'll want to set a locale once and forget about it. You can do that by including the locale in your application and writing to the instance variables that hold the underlying locale data for the SEGGER Embedded Studio C library.

For instance, you might wish to use Czech locale with a UTF codeset:

```
static __RAL_locale_t cz_locale =
{
  "cz_CZ.utf8",
  &__RAL_cs_CZ_locale,
  &__RAL_codeset_utf8
};
```

You can install this directly into the locale without using **setlocale**:

```
__RAL_global_locale.__category[LC_COLLATE]  = &cz_locale;
__RAL_global_locale.__category[LC_CTYPE]    = &cz_locale;
__RAL_global_locale.__category[LC_MONETARY] = &cz_locale;
__RAL_global_locale.__category[LC_NUMERIC]  = &cz_locale;
__RAL_global_locale.__category[LC_TIME]     = &cz_locale;
```

Complete API reference

This section contains a complete reference to the SEGGER Embedded Studio C library API.

File	Description
<assert.h>	Describes the diagnostic facilities which you can build into your application.
<debugio.h>	Describes the virtual console services and semi-hosting support that SEGGER Embedded Studio provides to help you when developing your applications.
<ctype.h>	Describes the character classification and manipulation functions.
<errno.h>	Describes the macros and error values returned by the C library.
<float.h>	Defines macros that expand to various limits and parameters of the standard floating point types.
<limits.h>	Describes the macros that define the extreme values of underlying C types.
<locale.h>	Describes support for localization specific settings.
<math.h>	Describes the mathematical functions provided by the C library.
<setjmp.h>	Describes the non-local goto capabilities of the C library.
<stdarg.h>	Describes the way in which variable parameter lists are accessed.
<stddef.h>	Describes standard type definitions.
<stdio.h>	Describes the formatted input and output functions.
<stdlib.h>	Describes the general utility functions provided by the C library.
<string.h>	Describes the string handling functions provided by the C library.
<time.h>	Describes the functions to get and manipulate date and time information provided by the C library.
<wchar.h>	Describes the facilities you can use to manipulate wide characters.

<assert.h>

API Summary

Macros	
assert	Allows you to place assertions and diagnostic tests into programs
Functions	
__assert	User defined behaviour for the assert macro

__assert

Synopsis

```
void __assert(const char *expression,
              const char *filename,
              int line);
```

Description

There is no default implementation of **__assert**. Keeping **__assert** out of the library means that you can can customize its behaviour without rebuilding the library. You must implement this function where **expression** is the stringized expression, **filename** is the filename of the source file and **line** is the linenumber of the failed assertion.

assert

Synopsis

```
#define assert(e) ...
```

Description

If **NDEBUG** is defined as a macro name at the point in the source file where **<assert.h>** is included, the **assert** macro is defined as:

```
#define assert(ignore) ((void)0)
```

If **NDEBUG** is not defined as a macro name at the point in the source file where **<assert.h>** is included, the **assert** macro expands to a **void** expression that calls **__assert**.

```
#define assert(e) ((e) ? (void)0 : __assert(#e, __FILE__, __LINE__))
```

When such an **assert** is executed and **e** is false, **assert** calls the **__assert** function with information about the particular call that failed: the text of the argument, the name of the source file, and the source line number. These are the stringized expression and the values of the preprocessing macros **__FILE__** and **__LINE__**.

Note

The **assert** macro is redefined according to the current state of **NDEBUG** each time that **<assert.h>** is included.

<complex.h>

API Summary

Trigonometric functions	
cacos	Compute inverse cosine of a complex float
cacosf	Compute inverse cosine of a complex float
casin	Compute inverse sine of a complex float
casinf	Compute inverse sine of a complex float
catan	Compute inverse tangent of a complex float
catanf	Compute inverse tangent of a complex float
ccos	Compute cosine of a complex float
ccosf	Compute cosine of a complex float
csin	Compute sine of a complex float
csinf	Compute sine of a complex float
ctan	Compute tangent of a complex float
ctanf	Compute tangent of a complex float
Hyperbolic trigonometric functions	
cacosh	Compute inverse hyperbolic cosine of a complex float
cacoshf	Compute inverse hyperbolic cosine of a complex float
casinh	Compute inverse hyperbolic sine of a complex float
casinhf	Compute inverse hyperbolic sine of a complex float
catanh	Compute inverse hyperbolic tangent of a complex float
catanhf	Compute inverse hyperbolic tangent of a complex float
ccosh	Compute hyperbolic cosine of a complex float
ccoshf	Compute hyperbolic cosine of a complex float
csinh	Compute hyperbolic sine of a complex float
csinhf	Compute hyperbolic sine of a complex float
ctanh	Compute hyperbolic tangent of a complex float
ctanhf	Compute hyperbolic tangent of a complex float
Exponential and logarithmic functions	
cexp	Computes the base-e exponential of a complex float
cexpf	Computes the base-e exponential of a complex float
clog	Computes the base-e logarithm of a complex float

clogf	Computes the base-e logarithm of a complex float
Power and absolute value functions	
cabs	Computes the absolute value of a complex float
cabsf	Computes the absolute value of a complex float
cpow	Compute a complex float raised to a power
cpowf	Compute a complex float raised to a power
csqrt	Compute square root of a complex float
csqrtf	Compute square root of a complex float
Manipulation functions	
carg	Compute argument of a complex float
cargf	Compute argument of a complex float
cimag	Compute imaginary part of a complex float
cimagf	Compute imaginary part of a complex float
conj	Compute conjugate of a complex float
conjf	Compute conjugate of a complex float
cproj	Compute projection on the Riemann sphere
cprojf	Compute projection on the Riemann sphere
creal	Compute real part of a complex float
crealf	Compute real part of a complex float

cabs

Synopsis

```
double cabs(double complex z);
```

Description

cabs returns the absolute value of **z**.

cabsf

Synopsis

```
float cabsf(float complex z);
```

Description

cabsf returns the absolute value of **z**.

cacos

Synopsis

```
double complex cacos(double complex z);
```

Description

cacos returns the principal value the inverse cosine of **z** with branch cuts outside the interval [-1,+1] on the real axis. The principal value lies in the interval [0,] on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

cacosf

Synopsis

```
float complex cacosf(float complex z);
```

Description

cacosf returns the principal value the inverse cosine of **z** with branch cuts outside the interval [-1,+1] on the real axis. The principal value lies in the interval [0,] on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

cacosh

Synopsis

```
double complex cacosh(double complex z);
```

Description

cacosh returns the principal value the inverse hyperbolic cosine of **z** with branch cuts of values less than 1 on the real axis. The principal value lies in the range of a half-strip of non-negative values on the real axis and in the interval [-i,+i] on the imaginary axis.

cacoshf

Synopsis

```
float complex cacoshf(float complex _z);
```

Description

cacoshf returns the principal value the inverse hyperbolic cosine of **z** with branch cuts of values less than 1 on the real axis. The principal value lies in the range of a half-strip of non-negative values on the real axis and in the interval [-i,+i] on the imaginary axis.

carg

Synopsis

```
double carg(double complex z);
```

Description

carg computes the argument of **z** with a branch cut along the negative real axis.

cargf

Synopsis

```
float cargf(float complex z);
```

Description

cargf computes the argument of **z** with a branch cut along the negative real axis.

casin

Synopsis

```
double complex casin(double complex z);
```

Description

casin returns the principal value the inverse sine of **z** with branch cuts outside the interval [-1,+1] on the real axis. The principal value lies in the interval [,] on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

casinf

Synopsis

```
float complex casinf(float complex z);
```

Description

casinf returns the principal value the inverse sine of **z** with branch cuts outside the interval [-1,+1] on the real axis. The principal value lies in the interval [,] on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

casinh

Synopsis

```
double complex casinh(double complex z);
```

Description

casinh returns the principal value the inverse hyperbolic sine of **z** with branch cuts outside the inteval [-i,+i] on the imaginary axis. The principal value lies in the range of a strip mathematically unbounded on the real axis and in the interval [-i,+i] on the imaginary axis.

casinhf

Synopsis

```
float complex casinhf(float complex z);
```

Description

casinhf returns the principal value the inverse hyperbolic sine of **z** with branch cuts outside the inteval [-i,+i] on the imaginary axis. The principal value lies in the range of a strip mathematically unbounded on the real axis and in the interval [-i,+i] on the imaginary axis.

catan

Synopsis

```
double complex catan(double complex z);
```

Description

catan returns the principal value the inverse sine of **z** with branch cuts outside the interval [-1,+1] on the real axis. The principal value lies in the interval [,] on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

catanf

Synopsis

```
float complex catanf(float complex z);
```

Description

catanf returns the principal value the inverse sine of **z** with branch cuts outside the interval [-1,+1] on the real axis. The principal value lies in the interval [,] on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

catanh

Synopsis

```
double complex catanh(double complex z);
```

Description

catanh returns the principal value the inverse hyperbolic sine of **z** with branch cuts outside the inteval [-1,+1] on the real axis. The principal value lies in the range of a strip mathematically unbounded on the real axis and in the interval [-i,+i] on the imaginary axis.

catanhf

Synopsis

```
float complex catanhf(float complex z);
```

Description

catanhf returns the principal value the inverse hyperbolic sine of **z** with branch cuts outside the inteval [-1,+1] on the real axis. The principal value lies in the range of a strip mathematically unbounded on the real axis and in the interval [-i,+i] on the imaginary axis.

ccos

Synopsis

```
double complex ccos(double complex z);
```

Description

ccos returns the complex cosine of **z**.

ccosf

Synopsis

```
float complex ccosf(float complex z);
```

Description

ccosf returns the complex cosine of **z**.

ccosh

Synopsis

```
double complex ccosh(double complex z);
```

Description

ccosh returns the complex hyperbolic cosine of **z**.

ccoshf

Synopsis

```
float complex ccoshf(float complex z);
```

Description

ccoshf returns the complex hyperbolic cosine of **z**.

cexp

Synopsis

```
double complex cexp(double complex z);
```

Description

cexp returns the complex base-e exponential value of **z**.

cexpf

Synopsis

```
float complex cexpf(float complex z);
```

Description

cexpf returns the complex base-e exponential value of **z**.

cimag

Synopsis

```
double cimag(double complex);
```

Description

cimag computes the imaginary part of **z**.

cimagf

Synopsis

```
float cimagf(float complex);
```

Description

cimagf computes the imaginary part of **z**.

clog

Synopsis

```
double complex clog(double complex z);
```

Description

clog returns the complex base-e logarithm value of **z**.

clogf

Synopsis

```
float complex clogf(float complex z);
```

Description

clogf returns the complex base-e logarithm value of **z**.

conj

Synopsis

```
double complex conj(double complex);
```

Description

conj computes the conjugate of **z** by reversing the sign of the imaginary part.

conjf

Synopsis

```
float complex conjf(float complex);
```

Description

conjf computes the conjugate of **z** by reversing the sign of the imaginary part.

cpow

Synopsis

```
double complex cpow(double complex x,
                    double complex y);
```

Description

cpow computes **x** raised to the power **y** with a branch cut for the **x** along the negative real axis.

cpowf

Synopsis

```
float complex cpowf(float complex x,
                    float complex y);
```

Description

cpowf computes **x** raised to the power **y** with a branch cut for the **x** along the negative real axis.

cproj

Synopsis

```
double complex cproj(double complex);
```

Description

cproj computes the projection of **z** on the Riemann sphere.

cprojf

Synopsis

```
float complex cprojf(float complex);
```

Description

cprojf computes the projection of **z** on the Riemann sphere.

creal

Synopsis

```
double creal(double complex);
```

Description

creal computes the real part of **z**.

crealf

Synopsis

```
float crealf(float complex);
```

Description

crealf computes the real part of **z**.

csin

Synopsis

```
double complex csin(double complex z);
```

Description

csin returns the complex sine of **z**.

csinf

Synopsis

```
float complex csinf(float complex z);
```

Description

csinf returns the complex sine of **z**.

csinh

Synopsis

```
double complex csinh(double complex z);
```

Description

csinh returns the complex hyperbolic sine of **z**.

csinhf

Synopsis

```
float complex csinhf(float complex z);
```

Description

csinhf returns the complex hyperbolic sine of **z**.

csqrt

Synopsis

```
double complex csqrt(double complex z);
```

Description

csqrt computes the complex square root of **z** with a branch cut along the negative real axis.

csqrtf

Synopsis

```
float complex csqrtf(float complex z);
```

Description

csqrtf computes the complex square root of **z** with a branch cut along the negative real axis.

ctan

Synopsis

```
double complex ctan(double complex z);
```

Description

ctan returns the complex tangent of **z**.

ctanf

Synopsis

```
float complex ctanf(float complex z);
```

Description

ctanf returns the complex tangent of **z**.

ctanh

Synopsis

```
double complex ctanh(double complex z);
```

Description

ctanh returns the complex hyperbolic tangent of **z**.

ctanhf

Synopsis

```
float complex ctanhf(float complex z);
```

Description

ctanhf returns the complex hyperbolic tangent of **z**.

<ctype.h>

API Summary

Classification functions	
isalnum	Is character alphanumeric?
isalpha	Is character alphabetic?
isblank	Is character a space or horizontal tab?
iscntrl	Is character a control?
isdigit	Is character a decimal digit?
isgraph	Is character any printing character except space?
islower	Is character a lowercase letter?
isprint	Is character printable?
ispunct	Is character a punctuation mark?
isspace	Is character a whitespace character?
isupper	Is character an uppercase letter?
isxdigit	Is character a hexadecimal digit?
Conversion functions	
tolower	Convert uppercase character to lowercase
toupper	Convert lowercase character to uppercase
Classification functions (extended)	
isalnum_l	Is character alphanumeric?
isalpha_l	Is character alphabetic?
isblank_l	Is character a space or horizontal tab?
iscntrl_l	Is character a control character?
isdigit_l	Is character a decimal digit?
isgraph_l	Is character any printing character except space?
islower_l	Is character a lowercase letter?
isprint_l	Is character printable?
ispunct_l	Is character a punctuation mark?
isspace_l	Is character a whitespace character?
isupper_l	Is character an uppercase letter?
isxdigit_l	Is character a hexadecimal digit?
Conversion functions (extended)	
tolower_l	Convert uppercase character to lowercase

toupper_l	Convert lowercase character to uppercase

isalnum

Synopsis

```
int isalnum(int c);
```

Description

isalnum returns nonzero (true) if and only if the value of the argument **c** is an alphabetic or numeric character.

isalnum_l

Synopsis

```
int isalnum_l(int c,
              locale_t loc);
```

Description

isalnum_l returns nonzero (true) if and only if the value of the argument **c** is a alphabetic or numeric character in locale **loc**.

isalpha

Synopsis

```
int isalpha(int c);
```

Description

isalpha returns true if the character **c** is alphabetic. That is, any character for which **isupper** or **islower** returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true.

In the C locale, **isalpha** returns nonzero (true) if and only if **isupper** or **islower** return true for value of the argument **c**.

isalpha_l

Synopsis

```
int isalpha_l(int c,
              locale_t loc);
```

Description

isalpha_l returns nonzero (true) if and only if **isupper** or **islower** return true for value of the argument **c** in locale **loc**.

isblank

Synopsis

```
int isblank(int c);
```

Description

isblank returns nonzero (true) if and only if the value of the argument **c** is either a space character (`' '`) or the horizontal tab character (`'\\t'`).

isblank_l

Synopsis

```
int isblank_l(int c,
              locale_t loc);
```

Description

isblank_l returns nonzero (true) if and only if the value of the argument **c** is either a space character (`' '`) or the horizontal tab character (`'\\t'`) in locale **loc**.

iscntrl

Synopsis

```
int iscntrl(int c);
```

Description

iscntrl returns nonzero (true) if and only if the value of the argument **c** is a control character. Control characters have values 0 through 31 and the single value 127.

iscntrl_l

Synopsis

```
int iscntrl_l(int c,
              locale_t loc);
```

Description

iscntrl_l returns nonzero (true) if and only if the value of the argument **c** is a control character in locale **loc**.

isdigit

Synopsis

```
int isdigit(int c);
```

Description

isdigit returns nonzero (true) if and only if the value of the argument **c** is a digit.

isdigit_l

Synopsis

```
int isdigit_l(int c,
              locale_t loc);
```

Description

isdigit_l returns nonzero (true) if and only if the value of the argument **c** is a decimal digit in locale **loc**.

isgraph

Synopsis

```
int isgraph(int c);
```

Description

isgraph returns nonzero (true) if and only if the value of the argument **c** is any printing character except space (`' '`).

isgraph_l

Synopsis

```
int isgraph_l(int c,
              locale_t loc);
```

Description

isgraph_l returns nonzero (true) if and only if the value of the argument **c** is any printing character except space (`' '`) in locale **loc**.

islower

Synopsis

```
int islower(int c);
```

Description

islower returns nonzero (true) if and only if the value of the argument **c** is an lowercase letter.

islower_l

Synopsis

```
int islower_l(int c,
              locale_t loc);
```

Description

islower_l returns nonzero (true) if and only if the value of the argument **c** is an lowercase letter in locale **loc**.

isprint

Synopsis

```
int isprint(int c);
```

Description

isprint returns nonzero (true) if and only if the value of the argument **c** is any printing character including space (`' '`).

isprint_l

Synopsis

```
int isprint_l(int c,
              locale_t loc);
```

Description

isprint_l returns nonzero (true) if and only if the value of the argument **c** is any printing character including space (`' '`) in locale **loc**.

ispunct

Synopsis

```
int ispunct(int c);
```

Description

ispunct returns nonzero (true) for every printing character for which neither **isspace** nor **isalnum** is true.

ispunct_l

Synopsis

```
int ispunct_l(int c,
              locale_t loc);
```

Description

ispunct_l returns nonzero (true) for every printing character for which neither **isspace** nor **isalnum** is true in in locale **loc**.

isspace

Synopsis

```
int isspace(int c);
```

Description

isspace returns nonzero (true) if and only if the value of the argument **c** is a standard white-space character. The standard white-space characters are space (`' '`), form feed (`'\\f'`), new-line (`'\\n'`), carriage return (`'\\r'`), horizontal tab (`'\\t'`), and vertical tab (`'\v'`).

isspace_l

Synopsis

```
int isspace_l(int c,
              locale_t loc);
```

Description

isspace_l returns nonzero (true) if and only if the value of the argument **c** is a standard white-space character in in locale **loc**..

isupper

Synopsis

```
int isupper(int c);
```

Description

isupper returns nonzero (true) if and only if the value of the argument **c** is an uppercase letter.

isupper_l

Synopsis

```
int isupper_l(int c,
              locale_t loc);
```

Description

isupper_l returns nonzero (true) if and only if the value of the argument **c** is an uppercase letter in locale **loc**.

isxdigit

Synopsis

```
int isxdigit(int c);
```

Description

isxdigit returns nonzero (true) if and only if the value of the argument **c** is a hexadecimal digit.

isxdigit_l

Synopsis

```
int isxdigit_l(int c,
               locale_t loc);
```

Description

isxdigit_l returns nonzero (true) if and only if the value of the argument **c** is a hexadecimal digit in locale **loc**.

tolower

Synopsis

```
int tolower(int c);
```

Description

tolower converts an uppercase letter to a corresponding lowercase letter. If the argument **c** is a character for which **isupper** is true and there are one or more corresponding characters, as specified by the current locale, for which **islower** is true, the **tolower** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

Note that even though **isupper** can return true for some characters, **tolower** may return that uppercase character unchanged as there are no corresponding lowercase characters in the locale.

tolower_l

Synopsis

```
int tolower_l(int c,
              locale_t loc);
```

Description

tolower_l converts an uppercase letter to a corresponding lowercase letter in locale **loc**. If the argument **c** is a character for which **isupper** is true in locale **loc**, **tolower_l** returns the corresponding lowercase letter; otherwise, the argument is returned unchanged.

toupper

Synopsis

```
int toupper(int c);
```

Description

toupper converts a lowercase letter to a corresponding uppercase letter. If the argument `is` a character for which **islower** is true and there are one or more corresponding characters, as specified by the current locale, for which **isupper** is true, **toupper** returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged. Note that even though **islower** can return true for some characters, **toupper** may return that lowercase character unchanged as there are no corresponding uppercase characters in the locale.

toupper_l

Synopsis

```
int toupper_l(int c,
              locale_t loc);
```

Description

toupper_l converts a lowercase letter to a corresponding uppercase letter in locale **loc**. If the argument **c** is a character for which **islower** is true in locale **loc**, **toupper_l** returns the corresponding uppercase letter; otherwise, the argument is returned unchanged.

<debugio.h>

API Summary

File Functions	
debug_clearerr	Clear error indicator
debug_fclose	Closes an open stream
debug_feof	Check end of file condition
debug_ferror	Check error indicator
debug_fflush	Flushes buffered output
debug_fgetc	Read a character from a stream
debug_fgetpos	Return file position
debug_fgets	Read a string
debug_filesize	Return the size of a file
debug_fopen	Opens a file on the host PC
debug_fprintf	Formatted write
debug_fprintf_c	Formatted write
debug_fputc	Write a character
debug_fputs	Write a string
debug_fread	Read data
debug_freopen	Reopens a file on the host PC
debug_fscanf	Formatted read
debug_fscanf_c	Formatted read
debug_fseek	Set file position
debug_fsetpos	Teturn file position
debug_ftell	Return file position
debug_fwrite	Write data
debug_remove	Deletes a file on the host PC
debug_rename	Renames a file on the host PC
debug_rewind	Set file position to the beginning
debug_tmpfile	Open a temporary file
debug_tmpnam	Generate temporary filename
debug_ungetc	Push a character
debug_vfprintf	Formatted write
debug_vfscanf	Formatted read

Debug Terminal Output Functions	
debug_printf	Formatted write
debug_printf_c	Formatted write
debug_putchar	Write a character
debug_puts	Write a string
debug_vprintf	Formatted write
Debug Terminal Input Functions	
debug_getch	Blocking character read
debug_getchar	Line-buffered character read
debug_getd	Line-buffered double read
debug_getf	Line-buffered float read
debug_geti	Line-buffered integer read
debug_getl	Line-buffered long read
debug_getll	Line-buffered long long read
debug_gets	String read
debug_getu	Line-buffered unsigned integer
debug_getul	Line-buffered unsigned long read
debug_getull	Line-buffered unsigned long long read
debug_kbhit	Polled character read
debug_scanf	Formatted read
debug_scanf_c	Formatted read
debug_vscanf	Formatted read
Debugger Functions	
debug_abort	Stop debugging
debug_break	Stop target
debug_enabled	Test if debug input/output is enabled
debug_exit	Stop debugging
debug_getargs	Get arguments
debug_loadsymbols	Load debugging symbols
debug_runtime_error	Stop and report error
debug_unloadsymbols	Unload debugging symbols
Misc Functions	
debug_getenv	Get environment variable value
debug_perror	Display error
debug_system	Execute command

debug_time	get time

debug_abort

Synopsis

```
void debug_abort(void);
```

Description

debug_abort causes the debugger to exit and a failure result is returned to the user.

debug_break

Synopsis

```
void debug_break(void);
```

Description

debug_break causes the debugger to stop the target and position the cursor at the line that called debug_break.

debug_clearerr

Synopsis

```
void debug_clearerr(DEBUG_FILE *stream);
```

Description

debug_clearerr clears any error indicator or end of file condition for the **stream**.

debug_enabled

Synopsis

```
int debug_enabled(void);
```

Description

debug_enabled returns non-zero if the debugger is connected - you can use this to test if a debug input/output functions will work.

debug_exit

Synopsis

```
void debug_exit(int result);
```

Description

debug_exit causes the debugger to exit and **result** is returned to the user.

debug_fclose

Synopsis

```
int debug_fclose(DEBUG_FILE *stream);
```

Description

debug_fclose flushes any buffered output of the **stream** and then closes the stream.

debug_fclose returns 0 on success or -1 if there was an error.

debug_feof

Synopsis

```
int debug_feof(DEBUG_FILE *stream);
```

Description

debug_feof returns non-zero if the end of file condition is set for the **stream**.

debug_ferror

Synopsis

```
int debug_ferror(DEBUG_FILE *stream);
```

Description

debug_ferror returns non-zero if the error indicator is set for the **stream**.

debug_fflush

Synopsis

```
int debug_fflush(DEBUG_FILE *stream);
```

Description

debug_fflush flushes any buffered output of the **stream**.

debug_fflush returns 0 on success or -1 if there was an error.

debug_fgetc

Synopsis

```
int debug_fgetc(DEBUG_FILE *stream);
```

Description

debug_fgetc reads and returns the next character on **stream** or -1 if no character is available.

debug_fgetpos

Synopsis

```
int debug_fgetpos(DEBUG_FILE *stream,
                  long *pos);
```

Description

debug_fgetpos is equivalent to **debug_fseek** .

debug_fgets

Synopsis

```
char *debug_fgets(char *s,
                  int n,
                  DEBUG_FILE *stream);
```

Description

debug_fgets reads at most **n**-1 characters or the characters up to (and including) a newline from the input **stream** into the array pointed to by **s**. A null character is written to the array after the input characters.

debug_fgets returns **s** on success, or 0 on error or end of file.

debug_filesize

Synopsis

```
int debug_filesize(DEBUG_FILE *stream);
```

Description

debug_filesize returns the size of the file associated with the **stream** in bytes.

debug_filesize returns -1 on error.

debug_fopen

Synopsis

```
DEBUG_FILE *debug_fopen(const char *filename,
                        const char *mode);
```

Description

debug_fopen opens the **filename** on the host PC and returns a stream or **0** if the open fails. The **filename** is a host PC filename which is opened relative to the debugger working directory. The **mode** is a string containing one of:

> **r** open file for reading.
> **w** create file for writing.
> **a** open or create file for writing and position at the end of the file.
> **r**+ open file for reading and writing.
> **w**+ create file for reading and writing.
> **a**+ open or create text file for reading and writing and position at the end of the file.

followed by one of:

> **t** for a text file.
> **b** for a binary file.

debug_fopen returns a stream that can be used to access the file or **0** if the open fails.

debug_fprintf

Synopsis

```
int debug_fprintf(DEBUG_FILE *stream,
                  const char *format,
                  ...);
```

Description

debug_fprintf writes to **stream**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. The **format** string is a standard C printf format string. The actual formatting is performed on the host by the debugger and therefore **debug_fprintf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_fprintf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

debug_fprintf_c

Synopsis

```
int debug_fprintf_c(DEBUG_FILE *stream,
                    __code const char *format,
                    ...);
```

Description

debug_fprintf_c is equivalent to **debug_fprintf** with the format string in code memory.

debug_fputc

Synopsis

```
int debug_fputc(int c,
                DEBUG_FILE *stream);
```

Description

debug_fputc writes the character **c** to the output **stream**.

debug_fputc returns the character written or -1 if an error occurred.

debug_fputs

Synopsis

```
int debug_fputs(const char *s,
                DEBUG_FILE *stream);
```

Description

debug_fputs writes the string pointed to by **s** to the output **stream** and appends a new-line character. The terminating null character is not written.

debug_fputs returns -1 if a write error occurs; otherwise it returns a nonnegative value.

debug_fread

Synopsis

```
int debug_fread(void *ptr,
                int size,
                int nobj,
                DEBUG_FILE *stream);
```

Description

debug_fread reads from the input **stream** into the array **ptr** at most **nobj** objects of size **size**.

debug_fread returns the number of objects read. If this number is different from **nobj** then **debug_feof** and **debug_ferror** can be used to determine status.

debug_freopen

Synopsis

```
DEBUG_FILE *debug_freopen(const char *filename,
                          const char *mode,
                          DEBUG_FILE *stream);
```

Description

debug_freopen is the same as **debug_open** except the file associated with the **stream** is closed and the opened file is then associated with the **stream**.

debug_fscanf

Synopsis

```
int debug_fscanf(DEBUG_FILE *stream,
                 const char *format,
                 ...);
```

Description

debug_fscanf reads from the input **stream**, under control of the string pointed to by **format**, that specifies how subsequent arguments are converted for input. The **format** string is a standard C scanf format string. The actual formatting is performed on the host by the debugger and therefore **debug_fscanf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_fscanf returns number of characters read, or a negative value if an output or encoding error occurred.

debug_fscanf_c

Synopsis

```
int debug_fscanf_c(DEBUG_FILE *stream,
                   __code const char *format,
                   ...);
```

Description

debug_fscanf_c is equivalent to **debug_fscanf** with the format string in code memory.

debug_fseek

Synopsis

```
int debug_fseek(DEBUG_FILE *stream,
                long offset,
                int origin);
```

Description

debug_fseek sets the file position for the **stream**. A subsequent read or write will access data at that position. The **origin** can be one of:

0 sets the position to **offset** bytes from the beginning of the file.

1 sets the position to **offset** bytes relative to the current position.

2 sets the position to **offset** bytes from the end of the file.

Note that for text files **offset** must be zero. **debug_fseek** returns zero on success, non-zero on error.

debug_fsetpos

Synopsis

```
int debug_fsetpos(DEBUG_FILE *stream,
                  const long *pos);
```

Description

debug_fsetpos is equivalent to **debug_fseek** with 0 as the **origin**.

debug_ftell

Synopsis

```
long debug_ftell(DEBUG_FILE *stream);
```

Description

debug_ftell returns the current file position of the **stream**.

debug_ftell returns -1 on error.

debug_fwrite

Synopsis

```
int debug_fwrite(const void *ptr,
                 int size,
                 int nobj,
                 DEBUG_FILE *stream);
```

Description

debug_fwrite write to the output **stream** from the array **ptr** at most **nobj** objects of size **size**.

debug_fwrite returns the number of objects written. If this number is different from **nobj** then **debug_feof** and **debug_ferror** can be used to determine status.

debug_getargs

Synopsis

```
int debug_getargs(unsigned bufsize,
                  unsigned char *buf);
```

Description

debug_getargs stores the debugger command line arguments into the memory pointed at by **buf** up to a maximum of **bufsize** bytes. The command line is stored as a C **argc** array of null terminated string and the number of entries is returned as the result.

debug_getch

Synopsis

```
int debug_getch(void);
```

Description

debug_getch reads one character from the Debug Terminal. This function will block until a character is available.

debug_getchar

Synopsis

```
int debug_getchar(void);
```

Description

debug_getchar reads one character from the **Debug Terminal**. This function uses line input and will therefore block until characters are available and ENTER has been pressed.

debug_getchar returns the character that has been read.

debug_getd

Synopsis

```
int debug_getd(double *);
```

Description

debug_getd reads a double from the **Debug Terminal**. The number is written to the double object pointed to by **d**.

debug_getd returns zero on success or -1 on error.

debug_getenv

Synopsis

```
char *debug_getenv(char *name);
```

Description

debug_getenv returns the value of the environment variable **name** or 0 if the environment variable cannot be found.

debug_getf

Synopsis

```
int debug_getf(float *f);
```

Description

debug_getf reads an float from the **Debug Terminal**. The number is written to the float object pointed to by **f**.

debug_getf returns zero on success or -1 on error.

debug_geti

Synopsis

```
int debug_geti(int *i);
```

Description

debug_geti reads an integer from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the integer object pointed to by **i**.

debug_geti returns zero on success or -1 on error.

debug_getl

Synopsis

```
int debug_getl(long *l);
```

Description

debug_getl reads a long from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the long object pointed to by **l**.

debug_getl returns zero on success or -1 on error.

debug_getll

Synopsis

```
int debug_getll(long long *ll);
```

Description

debug_getll reads a long long from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the long long object pointed to by **ll**.

debug_getll returns zero on success or -1 on error.

debug_gets

Synopsis

```
char *debug_gets(char *s);
```

Description

debug_gets reads a string from the Debug Terminal in memory pointed at by **s**. This function will block until ENTER has been pressed.

debug_gets returns the value of **s**.

debug_getu

Synopsis

```
int debug_getu(unsigned *u);
```

Description

debug_getu reads an unsigned integer from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the unsigned integer object pointed to by **u**.

debug_getu returns zero on success or -1 on error.

debug_getul

Synopsis

```
int debug_getul(unsigned long *ul);
```

Description

debug_getul reads an unsigned long from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the long object pointed to by **ul**.

debug_getul returns zero on success or -1 on error.

debug_getull

Synopsis

```
int debug_getull(unsigned long long *ull);
```

Description

debug_getull reads an unsigned long long from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the long long object pointed to by **ull**.

debug_getull returns zero on success or -1 on error.

debug_kbhit

Synopsis

```
int debug_kbhit(void);
```

Description

debug_kbhit polls the Debug Terminal for a character and returns a non-zero value if a character is available or 0 if not.

debug_loadsymbols

Synopsis

```
void debug_loadsymbols(const char *filename,
                       const void *address,
                       const char *breaksymbol);
```

Description

debug_loadsymbols instructs the debugger to load the debugging symbols in the file denoted by **filename**. The **filename** is a (macro expanded) host PC filename which is relative to the debugger working directory. The **address** is the load address which is required for debugging position independent executables, supply **NULL** for regular executables. The **breaksymbol** is the name of a symbol in the filename to set a temporary breakpoint on or **NULL**.

debug_perror

Synopsis

```
void debug_perror(const char *s);
```

Description

debug_perror displays the optional string **s** on the **Debug Terminal** together with a string corresponding to the errno value of the last Debug IO operation.

debug_printf

Synopsis

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

Description

debug_printf writes to the **Debug Terminal**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. The **format** string is a standard C printf format string. The actual formatting is performed on the host by the debugger and therefore **debug_printf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_printf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

debug_printf_c

Synopsis

```
int debug_printf_c(__code const char *format,
                   ...);
```

Description

debug_printf_c is equivalent to **debug_printf** with the format string in code memory.

debug_putchar

Synopsis

```
int debug_putchar(int c);
```

Description

debug_putchar write the character **c** to the Debug Terminal.

debug_putchar returns the character written or -1 if a write error occurs.

debug_puts

Synopsis

```
int debug_puts(const char *);
```

Description

debug_puts writes the string **s** to the Debug Terminal followed by a new-line character.

debug_puts returns -1 if a write error occurs, otherwise it returns a nonnegative value.

debug_remove

Synopsis

```
int debug_remove(const char *filename);
```

Description

debug_remove removes the filename denoted by **filename** and returns **0** on success or **-1** on error. The **filename** is a host PC filename which is relative to the debugger working directory.

debug_rename

Synopsis

```
int debug_rename(const char *oldfilename,
                 const char *newfilename);
```

Description

debug_rename renames the file denoted by **oldpath** to **newpath** and returns zero on success or non-zero on error. The **oldpath** and **newpath** are host PC filenames which are relative to the debugger working directory.

debug_rewind

Synopsis

```
void debug_rewind(DEBUG_FILE *stream);
```

Description

debug_rewind sets the current file position of the **stream** to the beginning of the file and clears any error and end of file conditions.

debug_runtime_error

Synopsis

```
void debug_runtime_error(const char *error);
```

Description

debug_runtime_error causes the debugger to stop the target, position the cursor at the line that called debug_runtime_error, and display the null-terminated string pointed to by **error**.

debug_scanf

Synopsis

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

Description

debug_scanf reads from the **Debug Terminal**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for input. The **format** string is a standard C scanf format string. The actual formatting is performed on the host by the debugger and therefore **debug_scanf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_scanf returns number of characters read, or a negative value if an output or encoding error occurred.

debug_scanf_c

Synopsis

```
int debug_scanf_c(__code const char *format,
                  ...);
```

Description

debug_scanf_c is equivalent to **debug_scanf** with the format string in code memory.

debug_system

Synopsis

```
int debug_system(char *command);
```

Description

debug_system executes the **command** with the host command line interpreter and returns the commands exit status.

debug_time

Synopsis

```
long debug_time(long *ptr);
```

Description

debug_time returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC), according to the system clock of the host computer. The return value is stored in ***ptr** if **ptr** is not NULL.

debug_tmpfile

Synopsis

```
DEBUG_FILE *debug_tmpfile(void);
```

Description

debug_tmpfile creates a temporary file on the host PC which is deleted when the stream is closed.

debug_tmpnam

Synopsis

```
char *debug_tmpnam(char *str);
```

Description

debug_tmpnam returns a unique temporary filename. If **str** is **NULL** then a static buffer is used to store the filename, otherwise the filename is stored in **str**. On success a pointer to the string is returned, on failure **0** is returned.

debug_ungetc

Synopsis

```
int debug_ungetc(int c,
                 DEBUG_FILE *stream);
```

Description

debug_ungetc pushes the character **c** onto the input **stream**. If successful **c** is returned, otherwise -1 is returned.

debug_unloadsymbols

Synopsis

```
void debug_unloadsymbols(const char *filename);
```

Description

debug_unloadsymbols instructs the debugger to unload the debugging symbols (previously loaded by a call to **debug_loadsymbols**) in the file denoted by **filename**. The **filename** is a host PC filename which is relative to the debugger working directory.

debug_vfprintf

Synopsis

```
int debug_vfprintf(DEBUG_FILE *stream,
                   const char *format,
                   __va_list);
```

Description

debug_vfprintf is equivalent to **debug_fprintf** with arguments passed using **stdarg**.h rather than a variable number of arguments.

debug_vfscanf

Synopsis

```
int debug_vfscanf(DEBUG_FILE *stream,
                  const char *format,
                  __va_list);
```

Description

debug_vfscanf is equivalent to **debug_fscanf** with arguments passed using **stdarg**.h rather than a variable number of arguments.

debug_vprintf

Synopsis

```
int debug_vprintf(const char *format,
                  __va_list);
```

Description

debug_vprintf is equivalent to **debug_printf** with arguments passed using **stdarg**.h rather than a variable number of arguments.

debug_vscanf

Synopsis

```
int debug_vscanf(const char *format,
                 __va_list);
```

Description

debug_vscanf is equivalent to **debug_scanf** with arguments passed using **stdarg**.h rather than a variable number of arguments.

<errno.h>

API Summary

Error numbers	
EDOM	Domain error
EILSEQ	Illegal byte sequence
EINVAL	Invalid argument
ENOMEM	No memory available
ERANGE	Result too large or too small
Macros	
errno	Last-set error condition

EDOM

Synopsis

```
#define EDOM   ...
```

Description

EDOM - an input argument is outside the defined domain of a mathematical function.

EILSEQ

Synopsis

```
#define EILSEQ   ...
```

Description

EILSEQ - A wide-character code has been detected that does not correspond to a valid character, or a byte sequence does not form a valid wide-character code.

EINVAL

Synopsis

```
#define EINVAL    0x06
```

Description

EINVAL - An argument was invalid, or a combination of arguments was invalid.

ENOMEM

Synopsis

```
#define ENOMEM    0x05
```

Description

ENOMEM - no memory can be allocated by a function in the library. Note that **malloc**, **calloc**, and **realloc** do not set **errno** to **ENOMEM** on failure, but other library routines (such as **duplocale**) may set **errno** to **ENOMEM** when memory allocation fails.

ERANGE

Synopsis

```
#define ERANGE   ...
```

Description

ERANGE - the result of the function is too large (overflow) or too small (underflow) to be represented in the available space.

errno

Synopsis

```
int errno;
```

Description

errno is treated as an writable l-value, but the implementation of how the l-value is read an written is hidden from the user.

The value of **errno** is zero at program startup, but is never set to zero by any library function. The value of **errno** may be set to a nonzero value by a library function, and this effect is documented in each function that does so.

Note

The ISO standard does not specify whether **errno** is a macro or an identifier declared with external linkage. Portable programs must not make assumptions about the implementation of **errno**.

In this implementation, **errno** expands to a function call to **__errno** (MSP430, AVR, MAXQ) or **__aeabi_errno_addr** (ARM) that returns a pointer to a volatile **int**. This function can be implemented by the application to provide a thread-specific **errno**.

<float.h>

API Summary

Double exponent minimum and maximum values	
DBL_MAX_10_EXP	The maximum exponent value in base 10 of a double
DBL_MAX_EXP	The maximum exponent value of a double
DBL_MIN_10_EXP	The minimal exponent value in base 10 of a double
DBL_MIN_EXP	The minimal exponent value of a double
Implementation	
DBL_DIG	The number of digits of precision of a double
DBL_MANT_DIG	The number of digits in a double
DECIMAL_DIG	The number of decimal digits that can be rounded without change
FLT_DIG	The number of digits of precision of a float
FLT_EVAL_METHOD	The evaluation format
FLT_MANT_DIG	The number of digits in a float
FLT_RADIX	The radix of the exponent representation
FLT_ROUNDS	The rounding mode
Float exponent minimum and maximum values	
FLT_MAX_10_EXP	The maximum exponent value in base 10 of a float
FLT_MAX_EXP	The maximum exponent value of a float
FLT_MIN_10_EXP	The minimal exponent value in base 10 of a float
FLT_MIN_EXP	The minimal exponent value of a float
Double minimum and maximum values	
DBL_EPSILON	The difference between 1 and the least value greater than 1 of a double
DBL_MAX	The maximum value of a double
DBL_MIN	The minimal value of a double
Float minimum and maximum values	
FLT_EPSILON	The difference between 1 and the least value greater than 1 of a float
FLT_MAX	The maximum value of a float
FLT_MIN	The minimal value of a float

DBL_DIG

Synopsis

```
#define DBL_DIG                                15
```

Description

DBL_DIG specifies The number of digits of precision of a **double**.

DBL_EPSILON

Synopsis

```
#define DBL_EPSILON 2.2204460492503131E-16
```

Description

DBL_EPSILON the minimum positive number such that 1.0 + DBL_EPSILON != 1.0.

DBL_MANT_DIG

Synopsis

```
#define DBL_MANT_DIG                    53
```

Description

DBL_MANT_DIG specifies the number of base **FLT_RADIX** digits in the mantissa part of a **double**.

DBL_MAX

Synopsis

```
#define DBL_MAX    1.7976931348623157E+308
```

Description

DBL_MAX is the maximum value of a **double**.

DBL_MAX_10_EXP

Synopsis

```
#define DBL_MAX_10_EXP                +308
```

Description

DBL_MAX_10_EXP is the maximum value in base 10 of the exponent part of a **double**.

DBL_MAX_EXP

Synopsis

```
#define DBL_MAX_EXP                         +1024
```

Description

DBL_MAX_EXP is the maximum value of base **FLT_RADIX** in the exponent part of a **double**.

DBL_MIN

Synopsis

```
#define DBL_MIN    2.2250738585072014E-308
```

Description

DBL_MIN is the minimum value of a **double**.

DBL_MIN_10_EXP

Synopsis

```
#define DBL_MIN_10_EXP                -307
```

Description

DBL_MIN_10_EXP is the minimum value in base 10 of the exponent part of a **double**.

DBL_MIN_EXP

Synopsis

```
#define DBL_MIN_EXP                       -1021
```

Description

DBL_MIN_EXP is the minimum value of base **FLT_RADIX** in the exponent part of a **double**.

DECIMAL_DIG

Synopsis

```
#define DECIMAL_DIG                17
```

Description

DECIMAL_DIG specifies the number of decimal digits that can be rounded to a floating-point number without change to the value.

FLT_DIG

Synopsis

```
#define FLT_DIG                    6
```

Description

FLT_DIG specifies The number of digits of precision of a **float**.

FLT_EPSILON

Synopsis

```
#define FLT_EPSILON 1.19209290E-07F // decimal constant
```

Description

FLT_EPSILON the minimum positive number such that 1.0 + FLT_EPSILON != 1.0.

FLT_EVAL_METHOD

Synopsis

```
#define FLT_EVAL_METHOD 0
```

Description

FLT_EVAL_METHOD specifies that all operations and constants are evaluated to the range and precision of the type.

FLT_MANT_DIG

Synopsis

```
#define FLT_MANT_DIG            24
```

Description

FLT_MANT_DIG specifies the number of base **FLT_RADIX** digits in the mantissa part of a **float**.

FLT_MAX

Synopsis

```
#define FLT_MAX     3.40282347E+38F
```

Description

FLT_MAX is the maximum value of a **float**.

FLT_MAX_10_EXP

Synopsis

```
#define FLT_MAX_10_EXP          +38
```

Description

FLT_MAX_10_EXP is the maximum value in base 10 of the exponent part of a **float**.

FLT_MAX_EXP

Synopsis

```
#define FLT_MAX_EXP             +128
```

Description

FLT_MAX_EXP is the maximum value of base **FLT_RADIX** in the exponent part of a **float**.

FLT_MIN

Synopsis

```
#define FLT_MIN     1.17549435E-38F
```

Description

FLT_MIN is the minimum value of a **float**.

FLT_MIN_10_EXP

Synopsis

```
#define FLT_MIN_10_EXP          -37
```

Description

FLT_MIN_10_EXP is the minimum value in base 10 of the exponent part of a **float**.

FLT_MIN_EXP

Synopsis

```
#define FLT_MIN_EXP            -125
```

Description

FLT_MIN_EXP is the minimum value of base **FLT_RADIX** in the exponent part of a **float**.

FLT_RADIX

Synopsis

```
#define FLT_RADIX                 2
```

Description

FLT_RADIX specifies the radix of the exponent representation.

FLT_ROUNDS

Synopsis

```
#define FLT_ROUNDS 1
```

Description

FLT_ROUNDS specifies the rounding mode of floating-point addition is round to nearest.

<iso646.h>

Overview

The header <iso646.h> defines macros that expand to the corresponding tokens to ease writing C programs with keyboards that do not have keys for frequently-used operators.

API Summary

Macros	
and	Alternative spelling for logical and operator
and_eq	Alternative spelling for logical and-equals operator
bitand	Alternative spelling for bitwise and operator
bitor	Alternative spelling for bitwise or operator
compl	Alternative spelling for bitwise complement operator
not	Alternative spelling for logical not operator
not_eq	Alternative spelling for not-equal operator
or	Alternative spelling for logical or operator
or_eq	Alternative spelling for bitwise or-equals operator
xor	Alternative spelling for bitwise exclusive or operator
xor_eq	Alternative spelling for bitwise exclusive-or-equals operator

and

Synopsis

```
#define and      &&
```

Description

and defines the alternative spelling for &&.

and_eq

Synopsis

```
#define and_eq  &=
```

Description

and_eq defines the alternative spelling for &=.

bitand

Synopsis

```
#define bitand  &
```

Description

bitand defines the alternative spelling for &.

bitor

Synopsis

```
#define bitor   |
```

Description

bitor defines the alternative spelling for **|**.

compl

Synopsis

```
#define compl   ~
```

Description

compl defines the alternative spelling for ~.

not

Synopsis

```
#define not     !
```

Description

not defines the alternative spelling for `!`.

not_eq

Synopsis

```
#define not_eq  !=
```

Description

not_eq defines the alternative spelling for !=.

or

Synopsis

```
#define or      ||
```

Description

or defines the alternative spelling for **||**.

or_eq

Synopsis

```
#define or_eq    |=
```

Description

or_eq defines the alternative spelling for **|=**.

xor

Synopsis

```
#define xor     ^
```

Description

xor defines the alternative spelling for ^.

xor_eq

Synopsis

```
#define xor_eq  ^=
```

Description

xor_eq defines the alternative spelling for ^=.

<limits.h>

API Summary

Long integer minimum and maximum values	
LONG_MAX	Maximum value of a long integer
LONG_MIN	Minimum value of a long integer
ULONG_MAX	Maximum value of an unsigned long integer
Character minimum and maximum values	
CHAR_MAX	Maximum value of a plain character
CHAR_MIN	Minimum value of a plain character
SCHAR_MAX	Maximum value of a signed character
SCHAR_MIN	Minimum value of a signed character
UCHAR_MAX	Maximum value of an unsigned char
Long long integer minimum and maximum values	
LLONG_MAX	Maximum value of a long long integer
LLONG_MIN	Minimum value of a long long integer
ULLONG_MAX	Maximum value of an unsigned long long integer
Short integer minimum and maximum values	
SHRT_MAX	Maximum value of a short integer
SHRT_MIN	Minimum value of a short integer
USHRT_MAX	Maximum value of an unsigned short integer
Integer minimum and maximum values	
INT_MAX	Maximum value of an integer
INT_MIN	Minimum value of an integer
UINT_MAX	Maximum value of an unsigned integer
Type sizes	
CHAR_BIT	Number of bits in a character
Multi-byte values	
MB_LEN_MAX	maximum number of bytes in a multi-byte character

CHAR_BIT

Synopsis

```
#define CHAR_BIT   8
```

Description

CHAR_BIT is the number of bits for smallest object that is not a bit-field (byte).

CHAR_MAX

Synopsis

```
#define CHAR_MAX   255
```

Description

CHAR_MAX is the maximum value for an object of type **char**.

CHAR_MIN

Synopsis

```
#define CHAR_MIN   0
```

Description

CHAR_MIN is the minimum value for an object of type **char**.

INT_MAX

Synopsis

```
#define INT_MAX    2147483647
```

Description

INT_MAX is the maximum value for an object of type **int**.

INT_MIN

Synopsis

```
#define INT_MIN    (-2147483647 - 1)
```

Description

INT_MIN is the minimum value for an object of type **int**.

LLONG_MAX

Synopsis

```
#define LLONG_MAX  9223372036854775807LL
```

Description

LLONG_MAX is the maximum value for an object of type **long long int**.

LLONG_MIN

Synopsis

```
#define LLONG_MIN  (-9223372036854775807LL - 1)
```

Description

LLONG_MIN is the minimum value for an object of type **long long int**.

LONG_MAX

Synopsis

```
#define LONG_MAX   2147483647L
```

Description

LONG_MAX is the maximum value for an object of type **long int**.

LONG_MIN

Synopsis

```
#define LONG_MIN    (-2147483647L - 1)
```

Description

LONG_MIN is the minimum value for an object of type **long int**.

MB_LEN_MAX

Synopsis

```
#define MB_LEN_MAX  4
```

Description

MB_LEN_MAX is the maximum number of bytes in a multi-byte character for any supported locale. Unicode (ISO 10646) characters between 0 and 10FFFF inclusive are supported which convert to a maximum of four bytes in the UTF-8 encoding.

SCHAR_MAX

Synopsis

```
#define SCHAR_MAX  127
```

Description

SCHAR_MAX is the maximum value for an object of type **signed char**.

SCHAR_MIN

Synopsis

```
#define SCHAR_MIN  (-128)
```

Description

SCHAR_MIN is the minimum value for an object of type **signed char**.

SHRT_MAX

Synopsis

```
#define SHRT_MAX   32767
```

Description

SHRT_MAX is the minimum value for an object of type **short int**.

SHRT_MIN

Synopsis

```
#define SHRT_MIN   (-32767 - 1)
```

Description

SHRT_MIN is the minimum value for an object of type **short int**.

UCHAR_MAX

Synopsis

```
#define UCHAR_MAX  255
```

Description

UCHAR_MAX is the maximum value for an object of type **unsigned char**.

UINT_MAX

Synopsis

```
#define UINT_MAX    4294967295U
```

Description

UINT_MAX is the maximum value for an object of type **unsigned int**.

ULLONG_MAX

Synopsis

```
#define ULLONG_MAX 18446744073709551615ULL
```

Description

ULLONG_MAX is the maximum value for an object of type **unsigned long long int**.

ULONG_MAX

Synopsis

```
#define ULONG_MAX  4294967295UL
```

Description

ULONG_MAX is the maximum value for an object of type **unsigned long int**.

USHRT_MAX

Synopsis

```
#define USHRT_MAX  65535
```

Description

USHRT_MAX is the minimum value for an object of type **unsigned short int**.

<locale.h>

API Summary

Structures	
lconv	Formatting info for numeric values
Functions	
localeconv	Get current locale data
setlocale	Set Locale

lconv

Synopsis

```
typedef struct {
  char *decimal_point;
  char *thousands_sep;
  char *grouping;
  char *int_curr_symbol;
  char *currency_symbol;
  char *mon_decimal_point;
  char *mon_thousands_sep;
  char *mon_grouping;
  char *positive_sign;
  char *negative_sign;
  char int_frac_digits;
  char frac_digits;
  char p_cs_precedes;
  char p_sep_by_space;
  char n_cs_precedes;
  char n_sep_by_space;
  char p_sign_posn;
  char n_sign_posn;
  char int_p_cs_precedes;
  char int_n_cs_precedes;
  char int_p_sep_by_space;
  char int_n_sep_by_space;
  char int_p_sign_posn;
  char int_n_sign_posn;
} lconv;
```

Description

lconv structure holds formatting information on how numeric values are to be written. Note that the order of fields in this structure is not consistent between implementations, nor is it consistent between C89 and C99 standards.

The members **decimal_point**, **grouping**, and **thousands_sep** are controlled by **LC_NUMERIC**, the remainder by **LC_MONETARY**.

The members **int_n_cs_precedes**, **int_n_sep_by_space**, **int_n_sign_posn**, **int_p_cs_precedes**, **int_p_sep_by_space**. and **int_p_sign_posn** are added by the C99 standard.

We have standardized on the ordering specified by the ARM EABI for the base of this structure. This ordering is neither that of C89 nor C99.

Member	Description
currency_symbol	Local currency symbol.
decimal_point	Decimal point separator.
frac_digits	Amount of fractional digits to the right of the decimal point for monetary quantities in the local format.

grouping	Specifies the amount of digits that form each of the groups to be separated by thousands_sep separator for non-monetary quantities.
int_curr_symbol	International currency symbol.
int_frac_digits	Amount of fractional digits to the right of the decimal point for monetary quantities in the international format.
mon_decimal_point	Decimal-point separator used for monetary quantities.
mon_grouping	Specifies the amount of digits that form each of the groups to be separated by mon_thousands_sep separator for monetary quantities.
mon_thousands_sep	Separators used to delimit groups of digits to the left of the decimal point for monetary quantities.
negative_sign	Sign to be used for negative monetary quantities.
n_cs_precedes	Whether the currency symbol should precede negative monetary quantities.
n_sep_by_space	Whether a space should appear between the currency symbol and negative monetary quantities.
n_sign_posn	Position of the sign for negative monetary quantities.
positive_sign	Sign to be used for nonnegative (positive or zero) monetary quantities.
p_cs_precedes	Whether the currency symbol should precede nonnegative (positive or zero) monetary quantities.
p_sep_by_space	Whether a space should appear between the currency symbol and nonnegative (positive or zero) monetary quantities.
p_sign_posn	Position of the sign for nonnegative (positive or zero) monetary quantities.
thousands_sep	Separators used to delimit groups of digits to the left of the decimal point for non-monetary quantities.

localeconv

Synopsis

```
localeconv(void);
```

Description

localeconv returns a pointer to a structure of type **lconv** with the corresponding values for the current locale filled in.

setlocale

Synopsis

```
char *setlocale(int category,
                const char *locale);
```

Description

setlocale sets the current locale. The **category** parameter can have the following values:

Name	Locale affected
LC_ALL	Entire locale
LC_COLLATE	Affects **strcoll** and **strxfrm**
LC_CTYPE	Affects character handling
LC_MONETARY	Affects monetary formatting information
LC_NUMERIC	Affects decimal-point character in I/O and string formatting operations
LC_TIME	Affects **strftime**

The **locale** parameter contains the name of a C locale to set or if **NULL** is passed the current locale is not changed.

Return Value

setlocale returns the name of the current locale.

<math.h>

API Summary

Comparison Macros	
isgreater	Is greater
isgreaterequal	Is greater or equal
isless	Is less
islessequal	Is less or equal
islessgreater	Is less or greater
isunordered	Is unordered
Classification Macros	
fpclassify	Classify floating type
isfinite	Test for a finite value
isinf	Test for infinity
isnan	Test for NaN
isnormal	Test for a normal value
signbit	Test sign
Trigonometric functions	
cos	Compute cosine of a double
cosf	Compute cosine of a float
sin	Compute sine of a double
sinf	Compute sine of a float
tan	Compute tangent of a double
tanf	Compute tangent of a double
Inverse trigonometric functions	
acos	Compute inverse cosine of a double
acosf	Compute inverse cosine of a float
asin	Compute inverse sine of a double
asinf	Compute inverse sine of a float
atan	Compute inverse tangent of a double
atan2	Compute inverse tangent of a ratio of doubles
atan2f	Compute inverse tangent of a ratio of floats
atanf	Compute inverse tangent of a float
Exponential and logarithmic functions	

exp	Compute exponential of a double
exp2	Compute binary exponential of a double
exp2f	Compute binary exponential of a float
expf	Compute exponential of a float
expm1	Compute exponential minus one of a double
expm1f	Compute exponential minus one of a float
frexp	Set exponent of a double
frexpf	Set exponent of a float
ilogb	Compute integer binary logarithm of a double
ilogbf	Compute integer binary logarithm of a float
ldexp	Adjust exponent of a double
ldexpf	Adjust exponent of a float
log	Compute natural logarithm of a double
log10	Compute common logarithm of a double
log10f	Compute common logarithm of a float
log1p	Compute natural logarithm plus one of a double
log1pf	Compute natural logarithm plus one of a float
log2	Compute binary logarithm of a double
log2f	Compute binary logarithm of a float
logb	Compute floating-point base logarithm of a double
logbf	Compute floating-point base logarithm of a float
logf	Compute natural logarithm of a float
scalbln	Scale a double
scalblnf	Scale a float
scalbn	Scale a double
scalbnf	Scale a float
Rounding and remainder functions	
ceil	Compute smallest integer not greater than a double
ceilf	Compute smallest integer not greater than a float
floor	Compute largest integer not greater than a double
floorf	Compute largest integer not greater than a float
fmod	Compute remainder after division of two doubles
fmodf	Compute remainder after division of two floats
llrint	Round and cast double to long long
llrintf	Round and cast float to long long

llround	Round and cast double to long long
llroundf	Round and cast float to long long
lrint	Round and cast double to long
lrintf	Round and cast float to long
lround	Round and cast double to long
lroundf	Round and cast float to long
modf	Break a double into integer and fractional parts
modff	Break a float into integer and fractional parts
nearbyint	Round double to nearby integral value
nearbyintf	Round float to nearby integral value
remainder	Compute remainder of a double
remainderf	Compute remainder of a float
remquo	Compute remainder and quotient of a double
remquof	Compute remainder and quotient of a float
rint	Round a double to an integral value
rintf	Round a float to an integral value
round	Round a double to the nearest integral value
roundf	Round a float to the nearest integral value
trunc	Truncate a double value
truncf	Truncate a float value
Power functions	
cbrt	Compute cube root of a double
cbrtf	Compute cube root of a float
hypot	Compute complex magnitude of two doubles
hypotf	Compute complex magnitude of two floats
pow	Raise a double to a power
powf	Raise a float to a power
sqrt	Compute square root of a double
sqrtf	Compute square root of a float
Absolute value functions	
fabs	Compute absolute value of a double
fabsf	Compute absolute value of a float
Maximum, minimum, and positive difference functions	
fdim	Compute positive difference of two doubles
fdimf	Compute positive difference of two floats

fmax	Compute maximum of two doubles
fmaxf	Compute maximum of two floats
fmin	Compute minimum of two doubles
fminf	Compute minimum of two floats
Hyperbolic functions	
cosh	Compute hyperbolic cosine of a double
coshf	Compute hyperbolic cosine of a float
sinh	Compute hyperbolic sine of a double
sinhf	Compute hyperbolic sine of a float
tanh	Compute hyperbolic tangent of a double
tanhf	Compute hyperbolic tangent of a float
Inverse hyperbolic functions	
acosh	Compute inverse hyperbolic cosine of a double
acoshf	Compute inverse hyperbolic cosine of a float
asinh	Compute inverse hyperbolic sine of a double
asinhf	Compute inverse hyperbolic sine of a float
atanh	Compute inverse hyperbolic tangent of a double
atanhf	Compute inverse hyperbolic tangent of a float
Fused multiply functions	
fma	Compute fused multiply-add of doubles
fmaf	Compute fused multiply-add of floats
Floating-point manipulation functions	
copysign	Copy magnitude and sign of a double
copysignf	Copy magnitude and sign of a float
nextafter	Next representable double value
nextafterf	Next representable float value
Error and Gamma functions	
erf	Compute error function of a double
erfc	Compute complementary error function of a double
erfcf	Compute complementary error function of a float
erff	Compute error function of a float
lgamma	Compute log-gamma function of a double
lgammaf	Compute log-gamma function of a float
tgamma	Compute gamma function of a double
tgammaf	Compute gamma function of a float

acos

Synopsis

```
double acos(double x);
```

Description

acos returns the principal value, in radians, of the inverse circular cosine of **x**. The principal value lies in the interval [0, PI] radians.

If |**x**| > 1, **errno** is set to **EDOM** and **acos** returns **HUGE_VAL**.

If **x** is NaN, **acos** returns **x**. If |**x**| > 1, **acos** returns NaN.

acosf

Synopsis

```
float acosf(float x);
```

Description

acosf returns the principal value, in radians, of the inverse circular cosine of **x**. The principal value lies in the interval [0, PI] radians.

If |**a**| 1, **errno** is set to **EDOM** and **acosf** returns **HUGE_VAL**.

If **x** is NaN, **acosf** returns **x**. If |**x**| > 1, **acosf** returns NaN.

acosh

Synopsis

```
double acosh(double x);
```

Description

acosh returns the non-negative inverse hyperbolic cosine of **x**.

acosh(*x*) is defined as **log**(*x* + **sqrt**(*x*^2 1)), assuming completely accurate computation.

If **x** < 1, **errno** is set to **EDOM** and **acosh** returns **HUGE_VAL**.

If **x** < 1, **acosh** returns NaN.
If **x** is NaN, **acosh** returns NaN.

acoshf

Synopsis

```
float acoshf(float x);
```

Description

acoshf returns the non-negative inverse hyperbolic cosine of **x**.

acosh(*x*) is defined as **log**(*x* + **sqrt**(*x*^2 1)), assuming completely accurate computation.

If **x** < 1, **errno** is set to **EDOM** and **acoshf** returns **HUGE_VALF**.

If **x** < 1, **acoshf** returns NaN.
If **x** is NaN, **acoshf** returns that NaN.

asin

Synopsis

```
double asin(double x);
```

Description

asin returns the principal value, in radians, of the inverse circular sine of **x**. The principal value lies in the interval [, +] radians.

If |**x**| > 1, **errno** is set to **EDOM** and **asin** returns **HUGE_VAL**.

If **x** is NaN, **asin** returns **x**. If |**x**| > 1, **asin** returns NaN.

asinf

Synopsis

```
float asinf(float x);
```

Description

asinf returns the principal value, in radians, of the inverse circular sine of **val**. The principal value lies in the interval [, +] radians.

If |**x**| > 1, **errno** is set to **EDOM** and **asinf** returns **HUGE_VALF**.

If **x** is NaN, **asinf** returns **x**. If |**x**| > 1, **asinf** returns NaN.

asinh

Synopsis

```
double asinh(double x);
```

Description

asinh calculates the hyperbolic sine of **x**.

If |**x**| > ~709.782, **errno** is set to **EDOM** and **asinh** returns **HUGE_VAL**.

If **x** is +, , or NaN, **asinh** returns |**x**|. If |**x**| > ~709.782, **asinh** returns + or depending upon the sign of **x**.

asinhf

Synopsis

```
float asinhf(float x);
```

Description

asinhf calculates the hyperbolic sine of **x**.

If |**x**| > ~88.7228, **errnois** set to **EDOM** and **asinhf** returns **HUGE_VALF**.

If **x** is +, , or NaN, **asinhf** returns |**x**|. If |**x**| > ~88.7228, **asinhf** returns + or depending upon the sign of **x**.

atan

Synopsis

```
double atan(double x);
```

Description

atan returns the principal value, in radians, of the inverse circular tangent of **x**. The principal value lies in the interval [, +] radians.

atan2

Synopsis

```
double atan2(double x,
             double y);
```

Description

atan2 returns the value, in radians, of the inverse circular tangent of **y** divided by **x** using the signs of **x** and **y** to compute the quadrant of the return value. The principal value lies in the interval [/2, +] radians. If **x** = **y** = 0, **errno** is set to **EDOM** and **atan2** returns **HUGE_VAL**.

atan2(**x**, NaN) is NaN.
atan2(NaN, **x**) is NaN.
atan2(0, +(anything but NaN)) is 0.
atan2(0, (anything but NaN)) is .
atan2((anything but 0 and NaN), 0) is .
atan2((anything but and NaN), +) is 0.
atan2((anything but and NaN),) is .
atan2(, +) is .
atan2(,) is .
atan2(, (anything but 0, NaN, and)) is .

atan2f

Synopsis

```
float atan2f(float y,
             float x);
```

Description

atan2f returns the value, in radians, of the inverse circular tangent of **y** divided by **x** using the signs of **x** and **y** to compute the quadrant of the return value. The principal value lies in the interval [, +] radians.

If **x** = **y** = 0, **errno** is set to **EDOM** and **atan2f** returns **HUGE_VALF**.

atan2f(**x**, NaN) is NaN.
atan2f(NaN, **x**) is NaN.
atan2f(0, +(anything but NaN)) is 0.
atan2f(0, (anything but NaN)) is .
atan2f((anything but 0 and NaN), 0) is .
atan2f((anything but and NaN), +) is 0.
atan2f((anything but and NaN),) is .
atan2f(, +) is .
atan2f(,) is .
atan2f(, (anything but 0, NaN, and)) is .

atanf

Synopsis

```
float atanf(float x);
```

Description

atanf returns the principal value, in radians, of the inverse circular tangent of **x**. The principal value lies in the interval [, +] radians.

atanh

Synopsis

```
double atanh(double x);
```

Description

atanh returns the inverse hyperbolic tangent of **x**.

If |**x**| 1, **errno** is set to **EDOM** and **atanh** returns **HUGE_VAL**.

If |**x**| > 1 **atanh** returns NaN.
If **x** is NaN, **atanh** returns that NaN.
If **x** is 1, **atanh** returns .
If **x** is 1, **atanh** returns .

atanhf

Synopsis

```
float atanhf(float x);
```

Description

atanhf returns the inverse hyperbolic tangent of **x**.

If |**x**| > 1 **atanhf** returns NaN. If **x** is NaN, **atanhf** returns that NaN. If **x** is 1, **atanhf** returns . If **x** is 1, **atanhf** returns .

cbrt

Synopsis

```
double cbrt(double x);
```

Description

cbrt computes the cube root of **x**.

cbrtf

Synopsis

```
float cbrtf(float x);
```

Description

cbrtf computes the cube root of **x**.

ceil

Synopsis

```
double ceil(double x);
```

Description

ceil computes the smallest integer value not less than **x**.

ceil (0) is 0. **ceil** ( ) is .

ceilf

Synopsis

```
float ceilf(float x);
```

Description

ceilf computes the smallest integer value not less than **x**.

ceilf (0) is 0. **ceilf** ( ) is .

copysign

Synopsis

```
double copysign(double x,
                double y);
```

Description

copysign returns a value with the magnitude of **x** and the sign of **y**.

copysignf

Synopsis

```
float copysignf(float x,
                float y);
```

Description

copysignf returns a value with the magnitude of **x** and the sign of **y**.

cos

Synopsis

```
double cos(double x);
```

Description

cos returns the radian circular cosine of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **cos** returns **HUGE_VAL**.

If **x** is NaN, **cos** returns **x**. If |**x**| is , **cos** returns NaN.

cosf

Synopsis

```
float cosf(float x);
```

Description

cosf returns the radian circular cosine of x.

If |**x**| > 10^9, **errno** is set to **EDOM** and **cosf** returns **HUGE_VALF**.

If **x** is NaN, **cosf** returns **x**. If |**x**| is , **cosf** returns NaN.

cosh

Synopsis

```
double cosh(double x);
```

Description

cosh calculates the hyperbolic cosine of **x**.

If |**x**| > ~709.782, **errno** is set to **EDOM** and **cosh** returns **HUGE_VAL**.

If **x** is +, , or NaN, **cosh** returns |**x**|.> If |**x**| > ~709.782, **cosh** returns + or depending upon the sign of **x**.

coshf

Synopsis

```
float coshf(float x);
```

Description

coshf calculates the hyperbolic sine of **x**.

If |**x**| > ~88.7228, **errno** is set to **EDOM** and **coshf** returns **HUGE_VALF**.

If **x** is +, , or NaN, **coshf** returns |**x**|.
If |**x**| > ~88.7228, **coshf** returns + or depending upon the sign of **x**.

erf

Synopsis

```
double erf(double x);
```

Description

erf returns the error function for **x**.

erfc

Synopsis

```
double erfc(double x);
```

Description

erfc returns the complementary error function for **x**.

erfcf

Synopsis

```
float erfcf(float x);
```

Description

erfcf returns the complementary error function for **x**.

erff

Synopsis

```
float erff(float x);
```

Description

erff returns the error function for **x**.

exp

Synopsis

```
double exp(double x);
```

Description

exp computes the base-*e* exponential of **x**.

If |**x**| > ~709.782, **errno** is set to **EDOM** and **exp** returns **HUGE_VAL**.

If **x** is NaN, **exp** returns NaN.
If **x** is , **exp** returns .
If **x** is , **exp** returns 0.

exp2

Synopsis

```
double exp2(double x);
```

Description

exp2 returns 2 raised to the power of **x**.

exp2f

Synopsis

```
float exp2f(float x);
```

Description

exp2f returns 2 raised to the power of **x**.

expf

Synopsis

```
float expf(float x);
```

Description

expf computes the base-*e* exponential of **x**.

If |**x**| > ~88.722, **errno** is set to **EDOM** and **expf** returns **HUGE_VALF**. If **x** is NaN, **expf** returns NaN.
If **x** is , **expf** returns .
If **x** is , **expf** returns 0.

expm1

Synopsis

```
double expm1(double x);
```

Description

expm1 returns *e* raised to the power of **x** minus one.

expm1f

Synopsis

```
float expm1f(float x);
```

Description

expm1f returns *e* raised to the power of **x** minus one.

fabs

Synopsis

```
double fabs(double x);
```

fabsf

Synopsis

```
float fabsf(float x);
```

Description

fabsf computes the absolute value of the floating-point number **x**.

fdim

Synopsis

```
double fdim(double x,
            double y);
```

Description

fdim returns the positive difference between **x** and **y**.

fdimf

Synopsis

```
float fdimf(float x,
            float y);
```

Description

fdimf returns the positive difference between **x** and **y**.

floor

Synopsis

```
double floor(double);
```

floor computes the largest integer value not greater than **x**.

floor (0) is 0. **floor** () is .

floorf

Synopsis

```
float floorf(float);
```

floorf computes the largest integer value not greater than **x**.

floorf(0) is 0. **floorf**() is .

fma

Synopsis

```
double fma(double x,
           double y,
           double z);
```

Description

fma computes **x y** + **z** with a single rounding.

fmaf

Synopsis

```
float fmaf(float x,
           float y,
           float z);
```

Description

fmaf computes **x** **y** + **z** with a single rounding.

fmax

Synopsis

```
double fmax(double x,
            double y);
```

Description

fmax determines the maximum of **x** and **y**.

fmax (NaN, **y**) is **y**. **fmax** (**x**, NaN) is **x**.

fmaxf

Synopsis

```
float fmaxf(float x,
            float y);
```

Description

fmaxf determines the maximum of **x** and **y**.

fmaxf (NaN, **y**) is **y**. **fmaxf**(**x**, NaN) is **x**.

fmin

Synopsis

```
double fmin(double x,
            double y);
```

Description

fmin determines the minimum of **x** and **y**.

fmin (NaN, **y**) is **y**. **fmin** (x, NaN) is **x**.

fminf

Synopsis

```
float fminf(float x,
            float y);
```

Description

fminf determines the minimum of **x** and **y**.

fminf (NaN, **y**) is **y**. **fminf** (**x**, NaN) is **x**.

fmod

Synopsis

```
double fmod(double x,
            double y);
```

Description

fmod computes the floating-point remainder of **x** divided by **y**. #b #this returns the value **x** *n* **y**, for some integer *n* such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**.

fmod (NaN, *y*) is NaN. **fmod** (*x*, NaN) is NaN. **fmod** (0, **y**) is 0 for **y** not zero.
fmod (, **y**) is NaN.
fmod (**x**, 0) is NaN.
fmod (x,) is **x** for **x** not infinite.

fmodf

Synopsis

```
float fmodf(float x,
            float y);
```

Description

fmodf computes the floating-point remainder of **x** divided by **y**. **fmodf** returns the value **x** *n* **y**, for some integer *n* such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**.

fmodf (NaN, *y*) is NaN. **fmodf** (*x*, NaN) is NaN. **fmodf** (0, **y**) is 0 for **y** not zero.
fmodf (, **y**) is NaN.
fmodf (**x**, 0) is NaN.
fmodf (x,) is **x** for **x** not infinite.

fpclassify

Synopsis

```
#define fpclassify(x) (__is_float32(x) ? __float32_classify(x) : __float64_classify(x))
```

Description

fpclassify classifies **x** as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. **fpclassify** returns one of:

FP_ZERO
FP_SUBNORMAL
FP_NORMAL
FP_INFINITE
FP_NAN

frexp

Synopsis

```
double frexp(double x,
             int *exp);
```

Description

frexp breaks a floating-point number into a normalized fraction and an integral power of 2.

frexp stores power of two in the **int** object pointed to by **exp** and returns the value **x**, such that **x** has a magnitude in the interval [1/2, 1) or zero, and value equals **x** * 2^**exp**.

If **x** is zero, both parts of the result are zero.

If **x** is or NaN, **frexp** returns **x** and stores zero into the int object pointed to by **exp**.

frexpf

Synopsis

```
float frexpf(float x,
             int *exp);
```

Description

frexpf breaks a floating-point number into a normalized fraction and an integral power of 2.

frexpf stores power of two in the **int** object pointed to by **frexpf** and returns the value **x**, such that **x** has a magnitude in the interval [, 1) or zero, and value equals **x** * 2^**exp**.

If **x** is zero, both parts of the result are zero.

If **x** is or NaN, **frexpf returns x** and stores zero into the int object pointed to by **exp**.

hypot

Synopsis

```
double hypot(double x,
             double y);
```

Description

hypot computes the square root of the sum of the squares of **x** and **y**, **sqrt**(**x*****x** + **y*****y**), without undue overflow or underflow. If **x** and **y** are the lengths of the sides of a right-angled triangle, then **hypot** computes the length of the hypotenuse.

If **x** or **y** is + or , **hypot** returns .
If **x** or **y** is NaN, **hypot** returns NaN.

hypotf

Synopsis

```
float hypotf(float x,
             float y);
```

Description

hypotf computes the square root of the sum of the squares of **x** and **y**, **sqrtf**(**x***x** + **y***y**), without undue overflow or underflow. If **x** and **y** are the lengths of the sides of a right-angled triangle, then **hypotf** computes the length of the hypotenuse.

If **x** or **y** is + or , **hypotf** returns . If **x** or **y** is NaN, **hypotf** returns NaN.

ilogb

Synopsis

```
int ilogb(double x);
```

Description

ilogb returns the integral part of the logarithm of **x**, using **FLT_RADIX** as the base for the logarithm.

ilogbf

Synopsis

```
int ilogbf(float x);
```

Description

ilogbf returns the integral part of the logarithm of **x**, using **FLT_RADIX** as the base for the logarithm.

isfinite

Synopsis

```
#define isfinite(x) (sizeof(x) == sizeof(float) ? __float32_isfinite(x) : __float64_isfinite(x))
```

Description

isfinite determines whether **x** is a finite value (zero, subnormal, or normal, and not infinite or NaN). **isfinite** returns a non-zero value if and only if **x** has a finite value.

isgreater

Synopsis

```
#define isgreater(x,y) (!isunordered(x, y) && (x > y))
```

Description

isgreater returns whether **x** is greater than **y**.

isgreaterequal

Synopsis

```
#define isgreaterequal(x,y) (!isunordered(x, y) && (x >= y))
```

Description

isgreaterequal returns whether **x** is greater than or equal to **y**.

isinf

Synopsis

```
#define isinf(x) (sizeof(x) == sizeof(float) ? __float32_isinf(x) : __float64_isinf(x))
```

Description

isinf determines whether **x** is an infinity (positive or negative). The determination is based on the type of the argument.

isless

Synopsis

```
#define isless(x,y) (!isunordered(x, y) && (x < y))
```

Description

isless returns whether **x** is less than **y**.

islessequal

Synopsis

```
#define islessequal(x,y) (!isunordered(x, y) && (x <= y))
```

Description

islessequal returns whether **x** is less than or equal to **y**.

islessgreater

Synopsis

```
#define islessgreater(x,y) (!isunordered(x, y) && (x < y || x > y))
```

Description

islessgreater returns whether **x** is less than or greater than **y**.

isnan

Synopsis

```
#define isnan(x) (sizeof(x) == sizeof(float) ? __float32_isnan(x) : __float64_isnan(x))
```

Description

isnan determines whether **x** is a NaN. The determination is based on the type of the argument.

isnormal

Synopsis

```
#define isnormal(x) (sizeof(x) == sizeof(float) ? __float32_isnormal(x) : __float64_isnormal(x))
```

Description

isnormal determines whether **x** is a normal value (zero, subnormal, or normal, and not infinite or NaN).. **isnormal** returns a non-zero value if and only if **x** has a normal value.

isunordered

Synopsis

```
#define isunordered(a,b) (fpclassify(a) == FP_NAN || fpclassify(b) == FP_NAN)
```

Description

isunordered returns whether **x** or **y** are unordered values.

ldexp

Synopsis

```
double ldexp(double x,
             int exp);
```

Description

ldexp multiplies a floating-point number by an integral power of 2.

ldexp returns **x** * 2^**exp**.

If the result overflows, **errno** is set to **ERANGE** and **ldexp** returns **HUGE_VALF**.

If **x** is or NaN, **ldexp** returns **x**. If the result overflows, **ldexp** returns .

ldexpf

Synopsis

```
float ldexpf(float x,
             int exp);
```

Description

ldexpf multiplies a floating-point number by an integral power of 2.

ldexpf returns **x** * 2^**exp**. If the result overflows, **errno** is set to **ERANGE** and **ldexpf** returns **HUGE_VALF**.

If **x** is or NaN, **ldexpf** returns **x**. If the result overflows, **ldexpf** returns .

lgamma

Synopsis

```
double lgamma(double x);
```

Description

lgamma returns the natural logarithm of the gamma function for **x**.

lgammaf

Synopsis

```
float lgammaf(float x);
```

Description

lgammaf returns the natural logarithm of the gamma function for **x**.

llrint

Synopsis

```
long long int llrint(double x);
```

Description

llrint rounds **x** to an integral value and returns it as a long long int.

llrintf

Synopsis

```
long long int llrintf(float x);
```

Description

llrintf rounds **x** to an integral value and returns it as a long long int.

llround

Synopsis

```
long long int llround(double x);
```

Description

llround rounds **x** to an integral value, with halfway cases rounded away from zero, and returns it as a long long int.

llroundf

Synopsis

```
long long int llroundf(float x);
```

Description

llroundf rounds **x** to an integral value, with halfway cases rounded away from zero, and returns it as a long long int.

log

Synopsis

```
double log(double x);
```

Description

log computes the base-*e* logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **log** returns **HUGE_VAL**. If **x** < 0, **errno** is set to **EDOM** and **log** returns **HUGE_VAL**.

If **x** < 0 or **x** = , **log** returns NaN.
If **x** = 0, **log** returns .
If **x** = , **log** returns .
If **x** = NaN, **log** returns **x**.

log10

Synopsis

```
double log10(double x);
```

Description

log10 computes the base-10 logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **log10** returns **HUGE_VAL**. If **x** < 0, **errno** is set to **EDOM** and **log10** returns **HUGE_VAL**.

If **x** < 0 or **x** = , **log10** returns NaN.
If **x** = 0, **log10** returns .
If **x** = , **log10** returns .
If **x** = NaN, **log10** returns **x**.

log10f

Synopsis

```
float log10f(float x);
```

Description

log10f computes the base-10 logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **log10f** returns **HUGE_VALF**. If **x** < 0, **errno** is set to **EDOM** and **log10f** returns **HUGE_VALF**.

If **x** < 0 or **x** = , **log10f** returns NaN.
If **x** = 0, **log10f** returns .
If **x** = , **log10f** returns .
If **x** = NaN, **log10f** returns **x**.

log1p

Synopsis

```
double log1p(double x);
```

Description

log1p computes the base-*e* logarithm of **x** plus one.

log1pf

Synopsis

```
float log1pf(float x);
```

Description

log1pf computes the base-*e* logarithm of **x** plus one.

log2

Synopsis

```
double log2(double x);
```

Description

log2 computes the base-*2* logarithm of **x**.

log2f

Synopsis

```
float log2f(float x);
```

Description

log2f computes the base-*2* logarithm of **x**.

logb

Synopsis

```
double logb(double x);
```

Description

logb computes the base-*FLT_RADIX* logarithm of **x**.

logbf

Synopsis

```
float logbf(float x);
```

Description

logbf computes the base-*FLT_RADIX* logarithm of **x**.

logf

Synopsis

```
float logf(float x);
```

Description

logf computes the base-*e* logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **logf** returns **HUGE_VALF**. If **x** < 0, **errno** is set to **EDOM** and **logf** returns **HUGE_VALF**.

If **x** < 0 or **x** = , **logf** returns NaN.
If **x** = 0, **logf** returns .
If **x** = , **logf** returns .
If **x** = NaN, **logf** returns **x**.

lrint

Synopsis

```
long int lrint(double x);
```

Description

lrint rounds **x** to an integral value and returns it as a long int.

lrintf

Synopsis

```
long int lrintf(float x);
```

Description

lrintf rounds **x** to an integral value and returns it as a long int.

lround

Synopsis

```
long int lround(double x);
```

Description

lround rounds **x** to an integral value, with halfway cases rounded away from zero, and returns it as a long int.

lroundf

Synopsis

```
long int lroundf(float x);
```

Description

lroundf rounds **x** to an integral value, with halfway cases rounded away from zero, and returns it as a long int.

modf

Synopsis

```
double modf(double x,
            double *iptr);
```

Description

modf breaks **x** into integral and fractional parts, each of which has the same type and sign as **x**.

The integral part (in floating-point format) is stored in the object pointed to by **iptr** and **modf** returns the signed fractional part of **x**.

modff

Synopsis

```
float modff(float x,
            float *iptr);
```

Description

modff breaks **x** into integral and fractional parts, each of which has the same type and sign as **x**.

The integral part (in floating-point format) is stored in the object pointed to by **iptr** and **modff** returns the signed fractional part of **x**.

nearbyint

Synopsis

```
double nearbyint(double);
```

Description

nearbyint Rounds **x** to an integral value.

nearbyintf

Synopsis

```
float nearbyintf(float);
```

Description

nearbyintf Rounds **x** to an integral value.

nextafter

Synopsis

```
double nextafter(double x,
                 double y);
```

Description

nextafter Returns the next representable value after **x** in the direction of **y**.

nextafterf

Synopsis

```
float nextafterf(float x,
                 float y);
```

Description

nextafterf Returns the next representable value after **x** in the direction of **y**.

pow

Synopsis

```
double pow(double x,
           double y);
```

Description

pow computes **x** raised to the power **y**.

If **x** < 0 and **y** 0, **errno** is set to **EDOM** and **pow** returns **HUGE_VAL**. If **x** 0 and **y** is not an integer value, **errno** is set to **EDOM** and **pow** returns **HUGE_VAL**.

If **y** = 0, **pow** returns 1.
If **y** = 1, **pow** returns **x**.
If **y** = NaN, **pow** returns NaN.
If **x** = NaN and **y** is anything other than 0, **pow** returns NaN.
If **x** < 1 or 1 < **x**, and **y** = +, **pow** returns +.
If **x** < 1 or 1 < **x**, and **y** = , **pow** returns 0.
If 1 < **x** < 1 and **y** = +, **pow** returns +0.
If 1 < **x** < 1 and **y** = , **pow** returns +.
If **x** = +1 or **x** = 1 and **y** = + or **y** = , **pow** returns NaN.
If **x** = +0 and **y** > 0 and **y** NaN, **pow** returns +0.
If **x** = 0 and **y** > 0 and **y** NaN or **y** not an odd integer, **pow** returns +0.
If **x** = +0 and **y** and **y** NaN, **pow** returns +.
If **x** = 0 and **y** > 0 and **y** NaN or **y** not an odd integer, **pow** returns +.
If **x** = 0 and **y** is an odd integer, **pow** returns 0.
If **x** = + and **y** > 0 and **y** NaN, **pow** returns +.
If **x** = + and **y** < 0 and **y** NaN, **pow** returns +0.
If **x** = , **pow** returns **pow**(0, **y**)
If **x** < 0 and **x** and **y** is a non-integer, **pow** returns NaN.

powf

Synopsis

```
float powf(float x,
           float y);
```

Description

powf computes **x** raised to the power **y**.

If **x** < 0 and **y** 0, **errno**. is set to **EDOM** and **powf** returns **HUGE_VALF**. If **x** 0 and **y** is not an integer value, **errno** is set to **EDOM** and **pow** returns **HUGE_VALF**.

If **y** = 0, **powf** returns 1.
If **y** = 1, **powf** returns **x**.
If **y** = NaN, **powf** returns NaN.
If **x** = NaN and **y** is anything other than 0, **powf** returns NaN.
If **x** < 1 or 1 < **x**, and **y** = +, **powf** returns +.
If **x** < 1 or 1 < **x**, and **y** = , **powf** returns 0.
If 1 < **x** < 1 and **y** = +, **powf** returns +0.
If 1 < **x** < 1 and **y** = , **powf** returns +.
If **x** = +1 or **x** = 1 and **y** = + or **y** = , **powf** returns NaN.
If **x** = +0 and **y** > 0 and **y** NaN, **powf** returns +0.
If **x** = 0 and **y** > 0 and **y** NaN or **y** not an odd integer, **powf** returns +0.
If **x** = +0 and **y** and **y** NaN, **powf** returns +.
If **x** = 0 and **y** > 0 and **y** NaN or **y** not an odd integer, **powf** returns +.
If **x** = 0 and **y** is an odd integer, **powf** returns 0.
If **x** = + and **y** > 0 and **y** NaN, **powf** returns +.
If **x** = + and **y** < 0 and **y** NaN, **powf** returns +0.
If **x** = , **powf** returns **powf**(0, **y**)
If **x** < 0 and **x** and **y** is a non-integer, **powf** returns NaN.

remainder

Synopsis

```
double remainder(double numer,
                 double denom);
```

Description

remainder computes the remainder of **numer** divided by **denom**.

remainderf

Synopsis

```
float remainderf(float numer,
                 float denom);
```

Description

remainderf computes the remainder of **numer** divided by **denom**.

remquo

Synopsis

```
double remquo(double numer,
              double denom,
              int *quot);
```

Description

remquo computes the remainder of **numer** divided by **denom** and the quotient pointed by **quot**.

remquof

Synopsis

```
float remquof(float numer,
              float denom,
              int *quot);
```

Description

remquof computes the remainder of **numer** divided by **denom** and the quotient pointed by **quot**.

rint

Synopsis

```
double rint(double x);
```

Description

rint rounds **x** to an integral value.

rintf

Synopsis

```
float rintf(float x);
```

Description

rintf rounds **x** to an integral value.

round

Synopsis

```
double round(double x);
```

Description

round rounds **x** to an integral value, with halfway cases rounded away from zero.

roundf

Synopsis

```
float roundf(float x);
```

Description

roundf rounds **x** to an integral value, with halfway cases rounded away from zero.

scalbln

Synopsis

```
double scalbln(double x,
               long int exp);
```

Description

scalbln multiplies **x** by **FLT_RADIX** raised to the power **exp**.

scalblnf

Synopsis

```
float scalblnf(float x,
               long int exp);
```

Description

scalblnf multiplies **x** by **FLT_RADIX** raised to the power **exp**.

scalbn

Synopsis

```
double scalbn(double x,
              int exp);
```

Description

scalbn multiplies a floating-point number by an integral power of **DBL_RADIX**.

As floating-point arithmetic conforms to IEC 60559, **DBL_RADIX** is 2 and **scalbn** is (in this implementation) identical to **ldexp**.

scalbn returns **x** * **DBL_RADIX^exp**.

If the result overflows, **errno** is set to **ERANGE** and **scalbn** returns **HUGE_VAL**.

If **x** is or NaN, **scalbn** returns **x**.
If the result overflows, **scalbn** returns .

See Also

ldexp

scalbnf

Synopsis

```
float scalbnf(float x,
              int exp);
```

Description

scalbnf multiplies a floating-point number by an integral power of **FLT_RADIX**.

As floating-point arithmetic conforms to IEC 60559, **FLT_RADIX** is 2 and **scalbnf** is (in this implementation) identical to **ldexpf**.

scalbnf returns **x** * **FLT_RADIX** ^**exp**.

If the result overflows, **errno** is set to **ERANGE** and **scalbnf** returns **HUGE_VALF**.

If **x** is or NaN, **scalbnf** returns **x**. If the result overflows, **scalbnf** returns .

See Also

ldexpf

signbit

Synopsis

```
#define signbit(x) (sizeof(x) == sizeof(float) ? __float32_signbit(x) : __float64_signbit(x))
```

Description

signbit macro determines whether the sign of **x** is negative. **signbit** returns a non-zero value if and only if **x** is negative.

sin

Synopsis

```
double sin(double x);
```

Description

sin returns the radian circular sine of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **sin** returns **HUGE_VAL**.

sin returns **x** if **x** is NaN. **sin** returns NaN if |**x**| is .

sinf

Synopsis

```
float sinf(float x);
```

Description

sinf returns the radian circular sine of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **sinf** returns **HUGE_VALF**.

sinf returns **x** if **x** is NaN. **sinf** returns NaN if |**x**| is .

sinh

Synopsis

```
double sinh(double x);
```

Description

sinh calculates the hyperbolic sine of **x**.

If |**x**| .782, **errno** is set to **EDOM** and **sinh** returns **HUGE_VAL**.

If **x** is +, , or NaN, **sinh** returns |**x**|. If |**x**| > ~709.782, **sinh** returns + or depending upon the sign of **x**.

sinhf

Synopsis

```
float sinhf(float x);
```

Description

sinhf calculates the hyperbolic sine of **x**.

If |**x**| > ~88.7228, **errno** is set to **EDOM** and **sinhf** returns **HUGE_VALF**.

If **x** is +, , or NaN, **sinhf** returns |**x**|. If |**x**| > ~88.7228, **sinhf** returns + or depending upon the sign of **x**.

sqrt

Synopsis

```
double sqrt(double x);
```

Description

sqrt computes the nonnegative square root of **x**. C90 and C99 require that a domain error occurs if the argument is less than zero sqrt deviates and always uses IEC 60559 semantics.

If **x** is +0, **sqrt** returns +0.
If **x** is 0, **sqrt** returns 0.
If **x** is , **sqrt** returns .
If **x** < 0, **sqrt** returns NaN.
If **x** is NaN, **sqrt** returns that NaN.

sqrtf

Synopsis

```
float sqrtf(float x);
```

Description

sqrtf computes the nonnegative square root of **x**. C90 and C99 require that a domain error occurs if the argument is less than zero sqrtf deviates and always uses IEC 60559 semantics.

If **x** is +0, **sqrtf** returns +0.
If **x** is 0, **sqrtf** returns 0.
If **x** is , **sqrtf** returns .
If **x** < 0, **sqrtf** returns NaN.
If **x** is NaN, **sqrtf** returns that NaN.

tan

Synopsis

```
double tan(double x);
```

Description

tan returns the radian circular tangent of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **tan** returns **HUGE_VAL**.

If **x** is NaN, **tan** returns **x**. If |**x**| is , **tan** returns NaN.

tanf

Synopsis

```
float tanf(float x);
```

Description

tanf returns the radian circular tangent of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **tanf** returns **HUGE_VALF**.

If **x** is NaN, tanf returns **x**. If |**x**| is , tanf returns NaN.

tanh

Synopsis

```
double tanh(double x);
```

Description

tanh calculates the hyperbolic tangent of **x**.

If **x** is NaN, **tanh** returns NaN.

tanhf

Synopsis

```
float tanhf(float x);
```

Description

tanhf calculates the hyperbolic tangent of **x**.

If **x** is NaN, **tanhf** returns NaN.

tgamma

Synopsis

```
double tgamma(double x);
```

Description

tgamma returns the gamma function for **x**.

tgammaf

Synopsis

```
float tgammaf(float x);
```

Description

tgammaf returns the gamma function for **x**.

trunc

Synopsis

```
double trunc(double x);
```

Description

trunc rounds **x** to an integral value that is not larger in magnitude than **x**.

truncf

Synopsis

```
float truncf(float x);
```

Description

truncf rounds **x** to an integral value that is not larger in magnitude than **x**.

<setjmp.h>

API Summary

Functions	
longjmp	Restores the saved environment
setjmp	Save calling environment for non-local jump

longjmp

Synopsis

```
void longjmp(jmp_buf env,
             int val);
```

Description

longjmp restores the environment saved by **setjmp** in the corresponding **env** argument. If there has been no such invocation, or if the function containing the invocation of **setjmp** has terminated execution in the interim, the behavior of **longjmp** is undefined.

After **longjmp** is completed, program execution continues as if the corresponding invocation of **setjmp** had just returned the value specified by **val**.

Note

longjmp cannot cause **setjmp** to return the value 0; if **val** is 0, **setjmp** returns the value 1.

Objects of automatic storage allocation that are local to the function containing the invocation of the corresponding **setjmp** that do not have **volatile** qualified type and have been changed between the **setjmp** invocation and **this** call are indeterminate.

setjmp

Synopsis

```
int setjmp(jmp_buf env);
```

Description

setjmp saves its calling environment in the **env** for later use by the **longjmp** function.

On return from a direct invocation **setjmp** returns the value zero. On return from a call to the **longjmp** function, the **setjmp** returns a nonzero value determined by the call to **longjmp**.

The environment saved by a call to **setjmp** consists of information sufficient for a call to the **longjmp** function to return execution to the correct block and invocation of that block, were it called recursively.

<stdarg.h>

API Summary

Macros	
va_arg	Get variable argument value
va_copy	Copy var args
va_end	Finish access to variable arguments
va_start	Start access to variable arguments

va_arg

Synopsis

```
type va_arg(va_list ap,
            type);
```

Description

va_arg expands to an expression that has the specified type and the value of the **type** argument. The **ap** parameter must have been initialized by **va_start** or **va_copy**, without an intervening invocation of **va_end**. You can create a pointer to a **va_list** and pass that pointer to another function, in which case the original function may make further use of the original list after the other function returns.

Each invocation of the **va_arg** macro modifies **ap** so that the values of successive arguments are returned in turn. The parameter type must be a type name such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to **type**.

If there is no actual next argument, or if type is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior of **va_arg** is undefined, except for the following cases:

> one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;
>
> one type is pointer to **void** and the other is a pointer to a character type.

The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by **parmN**. Successive invocations return the values of the remaining arguments in succession.

va_copy

Synopsis

```
void va_copy(va_list dest,
             val_list src);
```

Description

va_copy initializes **dest** as a copy of **src**, as if the **va_start** macro had been applied to **dest** followed by the same sequence of uses of the **va_arg** macro as had previously been used to reach the present state of **src**. Neither the **va_copy** nor **va_start** macro shall be invoked to reinitialize **dest** without an intervening invocation of the **va_end** macro for the same **dest**.

va_end

Synopsis

```
void va_end(va_list ap);
```

Description

va_end indicates a normal return from the function whose variable argument list **ap** was initialised by **va_start** or **va_copy**. The **va_end** macro may modify **ap** so that it is no longer usable without being reinitialized by **va_start** or **va_copy**. If there is no corresponding invocation of **va_start** or **va_copy**, or if **va_end** is not invoked before the return, the behavior is undefined.

va_start

Synopsis

```
void va_start(va_list ap,
              paramN);
```

Description

va_start initializes **ap** for subsequent use by the **va_arg** and **va_end** macros.

The parameter **parmN** is the identifier of the last fixed parameter in the variable parameter list in the function definition (the one just before the '**, ...').**

The behaviour of **va_start** and **va_arg** is undefined if the parameter **parmN** is declared with the **register** storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions.

va_start must be invoked before any access to the unnamed arguments.

va_start and **va_copy** must not be invoked to reinitialize **ap** without an intervening invocation of the **va_end** macro for the same **ap**.

<stddef.h>

API Summary

Macros	
NULL	NULL pointer
offsetof	offsetof
Types	
ptrdiff_t	ptrdiff_t type
size_t	size_t type

NULL

Synopsis

```
#define NULL 0
```

Description

NULL is the null pointer constant.

offsetof

Synopsis

```
#define offsetof(type, member)
```

Description

offsetof returns the offset in bytes to the structure **member**, from the beginning of its structure **type**.

ptrdiff_t

Synopsis

```
typedef __RAL_PTRDIFF_T ptrdiff_t;
```

Description

ptrdiff_t is the signed integral type of the result of subtracting two pointers.

size_t

Synopsis

```
typedef __RAL_SIZE_T size_t;
```

Description

size_t is the unsigned integral type returned by the sizeof operator.

<stdio.h>

API Summary

Character and string I/O functions	
getchar	Read a character from standard input
gets	Read a string from standard input
putchar	Write a character to standard output
puts	Write a string to standard output
Formatted output functions	
printf	Write formatted text to standard output
snprintf	Write formatted text to a string with truncation
sprintf	Write formatted text to a string
vprintf	Write formatted text to standard output using variable argument context
vsnprintf	Write formatted text to a string with truncation using variable argument context
vsprintf	Write formatted text to a string using variable argument context
Formatted input functions	
scanf	Read formatted text from standard input
sscanf	Read formatted text from string
vscanf	Read formatted text from standard using variable argument context
vsscanf	Read formatted text from a string using variable argument context

getchar

Synopsis

```
int getchar(void);
```

Description

getchar reads a single character from the standard input stream.

If the stream is at end-of-file or a read error occurs, **getchar** returns **EOF**.

gets

Synopsis

```
char *gets(char *s);
```

Description

gets reads characters from standard input into the array pointed to by **s** until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

gets returns **s** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and **gets** returns a null pointer. If a read error occurs during the operation, the array contents are indeterminate and **gets** returns a null pointer.

printf

Synopsis

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

Description

printf writes to the standard output stream using **putchar**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

printf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Formatted output control strings

The format is composed of zero or more directives: ordinary characters (not %, which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each conversion specification is introduced by the character %. After the % the following appear in sequence:

Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
An optional *minimum field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag has been given) to the field width. The field width takes the form of an asterisk * or a decimal integer.
An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, the number of digits to appear after the decimal-point character for e, E, f, and F conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of bytes to be written for s conversions. The precision takes the form of a period . followed either by an asterisk * or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
An optional length modifier that specifies the size of the argument.
A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an int argument supplies the field width or precision. The arguments specifying field width, or precision, or both, must appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

Some library variants do not support width and precision specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Width/Precision Support** property of the project if you use these.

Flag characters

The flag characters and their meanings are:

-

The result of the conversion is left-justified within the field. The default, if this flag is not specified, is that the result of the conversion is left-justified within the field.

+

The result of a signed conversion *always* begins with a plus or minus sign. The default, if this flag is not specified, is that it begins with a sign only when a negative value is converted.

space

If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and + flags both appear, the space flag is ignored.

#

The result is converted to an *alternative form*. For o conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both zero, a single 0 is printed). For x or X conversion, a nonzero result has 0x or 0X prefixed to it. For e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For g and F conversions, trailing zeros are not removed from the result. As an extension, when used in p conversion, the results has # prefixed to it. For other conversions, the behavior is undefined.

0

For d, i, o, u, x, X, e, E, f, F, g, and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.

Length modifiers

The length modifiers and their meanings are:

hh

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value will be converted to **signed char** or **unsigned char** before printing); or that a following n conversion specifier applies to a pointer to a **signed char** argument.

h

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value is converted to **short int** or **unsigned short int** before printing); or that a following n conversion specifier applies to a pointer to a **short int** argument.

l

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **long int** or **unsigned long int** argument; that a following n conversion specifier applies to a pointer to a **long int** argument; or has no effect on a following e, E, f, F, g, or G conversion specifier. Some library variants do not support the l length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

ll

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **long long int** or **unsigned long long int** argument; that a following n conversion specifier applies to a pointer to a **long long int** argument. Some library variants do not support the ll length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined. Note that the C99 length modifiers j, z, t, and L are not supported.

Conversion specifiers

The conversion specifiers and their meanings are:

d, i

The argument is converted to signed decimal in the style [-]*dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.

o, u, x, X

The unsigned argument is converted to unsigned octal for o, unsigned decimal for u, or unsigned hexadecimal notation for x or X in the style *dddd* the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.

f, F

A double argument representing a floating-point number is converted to decimal notation in the style [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. A double argument representing an infinity is converted to inf. A double argument representing a NaN is converted to nan. The F conversion specifier produces INF or NAN instead of inf or nan, respectively. Some library variants do not support the f and F conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.

e, E

A double argument representing a floating-point number is converted in the style [-]*d.dddedd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The E conversion specifier produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero. A double argument representing an infinity is converted to inf. A double argument representing a NaN is converted to nan. The E conversion specifier produces INF or NAN instead of inf or nan, respectively. Some library variants do not support the f and F conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf** Floating Point Support} property of the project if you use these conversion specifiers.

g, G

A double argument representing a floating-point number is converted in style f or e (or in style F or e in the case of a G conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. The style used depends on the value converted; style e (or E) is used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the # flag is specified; a decimal-point character appears only if it is followed by a digit. A double argument representing an infinity is converted to inf. A double argument representing a NaN is converted to nan. The G conversion specifier produces INF or NAN instead of inf or nan, respectively. Some library variants do not support the f and F conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.

c

The argument is converted to an **unsigned char**, and the resulting character is written.

s

The argument is be a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null character.

p

The argument is a pointer to **void**. The value of the pointer is converted in the same format as the x conversion specifier with a fixed precision of 2***sizeof**(**void** *).

n

The argument is a pointer to a signed integer into which is *written* the number of characters written to the output stream so far by the call to the formatting function. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

%

A % character is written. No argument is converted.

Note that the C99 width modifier l used in conjunction with the c and s conversion specifiers is not supported and nor are the conversion specifiers a and A.

putchar

Synopsis

```
int putchar(int c);
```

Description

putchar writes the character **c** to the standard output stream.

putchar returns the character written. If a write error occurs, **putchar** returns **EOF**.

puts

Synopsis

```
int puts(const char *s);
```

Description

puts writes the string pointed to by **s** to the standard output stream using **putchar** and appends a new-line character to the output. The terminating null character is not written.

puts returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

scanf

Synopsis

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

Description

scanf reads input from the standard input stream under control of the 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.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

scanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **scanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Formatted input control strings

The format is composed of zero or more directives: one or more white-space characters, an ordinary character (neither **%** nor a white-space character), or a conversion specification.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

> An optional assignment-suppressing character *.
> An optional nonzero decimal integer that specifies the maximum field width (in characters).
> An optional length modifier that specifies the size of the receiving object.
> A conversion specifier character that specifies the type of conversion to be applied.

The formatted input function executes each directive of the format in turn. If a directive fails, the function returns. Failures are described as input failures (because of the occurrence of an encoding error or the unavailability of input characters), or matching failures (because of inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.
An input item is read from the stream, unless the specification includes an n specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.
Except in the case of a **%** specifier, the input item (or, in the case of a %n directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

Length modifiers

The length modifiers and their meanings are:

hh

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **signed char** or pointer to **unsigned char**.

h

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

l

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to **double**. Some library variants do not support the l length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

ll

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**. Some library variants do not support the ll length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined. Note that the C99 length modifiers j, z, t, and L are not supported.

Conversion specifiers

d

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument must be a pointer to signed integer.

i

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value zero for the **base** argument. The corresponding argument must be a pointer to signed integer.

o

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 18 for the **base** argument. The corresponding argument must be a pointer to signed integer.

u

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument must be a pointer to unsigned integer.

x

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument must be a pointer to unsigned integer.

e, f, g

Matches an optionally signed floating-point number whose format is the same as expected for the subject sequence of the **strtod** function. The corresponding argument shall be a pointer to floating. Some library variants do not support the e, f and F conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Scanf Floating Point Support** property of the project if you use these conversion specifiers.

c

Matches a sequence of characters of exactly the number specified by the field width (one if no field width is present in the directive). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

s

Matches a sequence of non-white-space characters The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

[

Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket]. The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex ^, in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with [] or[^], the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a - character is in the scanlist and is not the first, nor the second where the first character is a ^, nor the last character, it is treated as a member of the scanset. Some library variants do not support the [conversion specifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Scanf Classes Supported** property of the project if you use this conversion specifier.

p

Reads a sequence output by the corresponding %p formatted output conversion. The corresponding argument must be a pointer to a pointer to **void**.

n

No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the formatted input function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the fscanf function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

%

Matches a single % character; no conversion or assignment occurs.

Note that the C99 width modifier l used in conjunction with the c, s, and [conversion specifiers is not supported and nor are the conversion specifiers a and A.

snprintf

Synopsis

```
int snprintf(char *s,
             size_t n,
             const char *format,
             ...);
```

Description

snprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output.

If **n** is zero, nothing is written, and **s** can be a null pointer. Otherwise, output characters beyond the **n**1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

snprintf returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**>.

sprintf

Synopsis

```
int sprintf(char *s,
            const char *format,
            ...);
```

Description

sprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. A null character is written at the end of the characters written; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

sprintf returns number of characters transmitted (not counting the terminating null), or a negative value if an output or encoding error occurred.

sscanf

Synopsis

```
int sscanf(const char *s,
           const char *format,
           ...);
```

Description

sscanf reads input from the string **s** under control of the 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.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

sscanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **sscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

vprintf

Synopsis

```
int vprintf(const char *format,
            __va_list arg);
```

Description

vprintf writes to the standard output stream using **putchar** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vprintf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vprintf** does not invoke the **va_end** macro.

vprintf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Note

vprintf is equivalent to **printf** with the variable argument list replaced by **arg**.

vscanf

Synopsis

```
int vscanf(const char *format,
           __va_list arg);
```

Description

vscanf reads input from the standard input stream under control of the 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. Before calling **vscanf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vscanf** does not invoke the **va_end** macro.

If there are insufficient arguments for the format, the behavior is undefined.

vscanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **vscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Note

vscanf is equivalent to **scanf** with the variable argument list replaced **arg**.

vsnprintf

Synopsis

```
int vsnprintf(char *s,
              size_t n,
              const char *format,
              __va_list arg);
```

Description

vsnprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vsnprintf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vsnprintf** does not invoke the **va_end** macro.

If **n** is zero, nothing is written, and **s** can be a null pointer. Otherwise, output characters beyond the **n**1^{st} are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

vsnprintf returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

Note

vsnprintf is equivalent to **snprintf** with the variable argument list replaced by **arg**.

vsprintf

Synopsis

```
int vsprintf(char *s,
             const char *format,
             __va_list arg);
```

Description

vsprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vsprintf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vsprintf** does not invoke the **va_end** macro.

A null character is written at the end of the characters written; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

vsprintf returns number of characters transmitted (not counting the terminating null), or a negative value if an output or encoding error occurred.

Note

vsprintf is equivalent to **sprintf** with the variable argument list replaced by **arg**.

vsscanf

Synopsis

```
int vsscanf(const char *s,
            const char *format,
            __va_list arg);
```

Description

vsscanf reads input from the string **s** under control of the 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. Before calling **vsscanf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vsscanf** does not invoke the **va_end** macro.

If there are insufficient arguments for the format, the behavior is undefined.

vsscanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **vsscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Note

vsscanf is equivalent to **sscanf** with the variable argument list replaced by **arg**.

<stdlib.h>

API Summary

Macros	
EXIT_FAILURE	EXIT_FAILURE
EXIT_SUCCESS	EXIT_SUCCESS
MB_CUR_MAX	Maximum number of bytes in a multi-byte character in the current locale
RAND_MAX	RAND_MAX
Types	
div_t	Structure containing quotient and remainder after division of an int
ldiv_t	Structure containing quotient and remainder after division of a long
lldiv_t	Structure containing quotient and remainder after division of a long long
Integer arithmetic functions	
abs	Return an integer absolute value
div	Divide two ints returning quotient and remainder
labs	Return a long integer absolute value
ldiv	Divide two longs returning quotient and remainder
llabs	Return a long long integer absolute value
lldiv	Divide two long longs returning quotient and remainder
Memory allocation functions	
calloc	Allocate space for an array of objects and initialize them to zero
free	Frees allocated memory for reuse
malloc	Allocate space for a single object
realloc	Resizes allocated memory space or allocates memory space
String to number conversions	
atof	Convert string to double
atoi	Convert string to int
atol	Convert string to long
atoll	Convert string to long long

strtod	Convert string to double
strtof	Convert string to float
strtol	Convert string to long
strtoll	Convert string to long long
strtoul	Convert string to unsigned long
strtoull	Convert string to unsigned long long
Pseudo-random sequence generation functions	
rand	Return next random number in sequence
srand	Set seed of random number sequence
Search and sort functions	
bsearch	Search a sorted array
qsort	Sort an array
Environment	
atexit	Set function to be execute on exit
exit	Terminates the calling process
Number to string conversions	
itoa	Convert int to string
lltoa	Convert long long to string
ltoa	Convert long to string
ulltoa	Convert unsigned long long to string
ultoa	Convert unsigned long to string
utoa	Convert unsigned to string
Multi-byte/wide character conversion functions	
mblen	Determine number of bytes in a multi-byte character
mblen_l	Determine number of bytes in a multi-byte character
Multi-byte/wide string conversion functions	
mbstowcs	Convert multi-byte string to wide string
mbstowcs_l	Convert multi-byte string to wide string using specified locale
mbtowc	Convert multi-byte character to wide character
mbtowc_l	Convert multi-byte character to wide character

EXIT_FAILURE

Synopsis

```
#define EXIT_FAILURE    1
```

Description

EXIT_FAILURE pass to **exit** on unsuccessful termination.

EXIT_SUCCESS

Synopsis

```
#define EXIT_SUCCESS    0
```

Description

EXIT_SUCCESS pass to **exit** on successful termination.

MB_CUR_MAX

Synopsis

```
#define MB_CUR_MAX  __RAL_mb_max(&__RAL_global_locale)
```

Description

MB_CUR_MAX expands to a positive integer expression with type **size_t** that is the maximum number of bytes in a multi-byte character for the extended character set specified by the current locale (category LC_CTYPE). **MB_CUR_MAX** is never greater than **MB_LEN_MAX**.

RAND_MAX

Synopsis

```
#define RAND_MAX 32767
```

Description

RAND_MAX expands to an integer constant expression that is the maximum value returned by **rand**.

abs

Synopsis

```
int abs(int j);
```

Description

abs returns the absolute value of the integer argument **j**.

atexit

Synopsis

```
int atexit(void (*func)(void));
```

Description

atexit registers **function** to be called when the application has exited. The functions registered with **atexit** are executed in reverse order of their registration. **atexit** returns 0 on success and non-zero on failure.

atof

Synopsis

```
double atof(const char *nptr);
```

Description

atof converts the initial portion of the string pointed to by **nptr** to a **double** representation.

atof does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atof** is equivalent to `strtod(nptr, (char **)NULL)`.

atof returns the converted value.

See Also

strtod

atoi

Synopsis

```
int atoi(const char *nptr);
```

Description

atoi converts the initial portion of the string pointed to by **nptr** to an **int** representation.

atoi does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atoi** is equivalent to `(int)strtol(nptr, (char **)NULL, 10)`.

atoi returns the converted value.

See Also

strtol

atol

Synopsis

```
long int atol(const char *nptr);
```

Description

atol converts the initial portion of the string pointed to by **nptr to** a **long int** representation.

atol does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atol** is equivalent to `strtol(nptr, (char **)NULL, 10)`.

atol returns the converted value.

See Also

strtol

atoll

Synopsis

```
long long int atoll(const char *nptr);
```

Description

atoll converts the initial portion of the string pointed to by **nptr** to a **long long int** representation.

atoll does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atoll** is equivalent to `strtoll(nptr, (char **)NULL, 10)`.

atoll returns the converted value.

See Also

strtoll

bsearch

Synopsis

```
void *bsearch(const void *key,
              const void *buf,
              size_t num,
              size_t size,
              int (*compare)(const void *, const void *));
```

Description

bsearch searches the array ***base** for the specified ***key** and returns a pointer to the first entry that matches or null if no match. The array should have **num** elements of **size** bytes and be sorted by the same algorithm as the **compare** function.

The **compare** function should return a negative value if the first parameter is less than second parameter, zero if the parameters are equal, and a positive value if the first parameter is greater than the second parameter.

calloc

Synopsis

```
void *calloc(size_t nobj,
             size_t size);
```

Description

calloc allocates space for an array of **nmemb** objects, each of whose size is **size**. The space is initialized to all zero bits.

calloc returns a null pointer if the space for the array of object cannot be allocated from free memory; if space for the array can be allocated, **calloc** returns a pointer to the start of the allocated space.

div

Synopsis

```
div_t div(int numer,
          int denom);
```

Description

div computes **numer** / **denom** and **numer** % **denom** in a single operation.

div returns a structure of type **div_t** comprising both the quotient and the remainder. The structures contain the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

See Also

div_t

div_t

Description

div_t stores the quotient and remainder returned by **div**.

exit

Synopsis

```
void exit(int exit_code);
```

Description

exit returns to the startup code and performs the appropriate cleanup process.

free

Synopsis

```
void free(void *p);
```

Description

free causes the space pointed to by **ptr** to be deallocated, that is, made available for further allocation. If **ptr** is a null pointer, no action occurs.

If **ptr** does not match a pointer earlier returned by **calloc**, **malloc**, or **realloc**, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

itoa

Synopsis

```
char *itoa(int val,
           char *buf,
           int radix);
```

Description

itoa converts **val** to a string in base **radix** and places the result in **buf**.

itoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and **radix** is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

See Also

ltoa, **lltoa**, **ultoa**, **ulltoa**, **utoa**

labs

Synopsis

```
long int labs(long int j);
```

Description

labs returns the absolute value of the long integer argument **j**.

ldiv

Synopsis

```
ldiv_t ldiv(long int numer,
            long int denom);
```

Description

ldiv computes **numer** / **denom** and **numer** % **denom** in a single operation.

ldiv returns a structure of type **ldiv_t** comprising both the quotient and the remainder. The structures contain the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

See Also

ldiv_t

ldiv_t

Description

ldiv_t stores the quotient and remainder returned by **ldiv**.

llabs

Synopsis

```
long long int llabs(long long int j);
```

Description

llabs returns the absolute value of the long long integer argument **j**.

lldiv

Synopsis

```
lldiv_t lldiv(long long int numer,
              long long int denom);
```

lldiv computes **numer** / **denom** and **numer** % **denom** in a single operation.

lldiv returns a structure of type **lldiv_t** comprising both the quotient and the remainder. The structures contain the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

See Also

lldiv_t

lldiv_t

Description

lldiv_t stores the quotient and remainder returned by **lldiv**.

lltoa

Synopsis

```
char *lltoa(long long val,
            char *buf,
            int radix);
```

Description

lltoa converts **val** to a string in base **radix** and places the result in **buf**.

lltoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and radix is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

See Also

itoa, **ltoa**, **ultoa**, **ulltoa**, **utoa**

ltoa

Synopsis

```
char *ltoa(long val,
           char *buf,
           int radix);
```

Description

ltoa converts **val** to a string in base **radix** and places the result in **buf**.

ltoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and radix is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

See Also

itoa, **lltoa**, **ultoa**, **ulltoa**, **utoa**

malloc

Synopsis

```
void *malloc(size_t size);
```

Description

malloc allocates space for an object whose size is specified by 'b size and whose value is indeterminate.

malloc returns a null pointer if the space for the object cannot be allocated from free memory; if space for the object can be allocated, **malloc** returns a pointer to the start of the allocated space.

mblen

Synopsis

```
int mblen(const char *s,
          size_t n);
```

Description

mblen determines the number of bytes contained in the multi-byte character pointed to by **s** in the current locale.

If **s** is a null pointer, **mblen** returns a nonzero or zero value, if multi-byte character encodings, respectively, do or do not have state-dependent encodings

If **s** is not a null pointer, **mblen** either returns 0 (if **s** points to the null character), or returns the number of bytes that are contained in the multi-byte character (if the next **n** or fewer bytes form a valid multi-byte character), or returns 1 (if they do not form a valid multi-byte character).

Note

Except that the conversion state of the **mbtowc** function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, s, n);
```

Note

It is guaranteed that no library function in the Standard C library calls **mblen**.

See Also

mblen_l, **mbtowc**

mblen_l

Synopsis

```
int mblen_l(const char *s,
            size_t n,
            __locale_s *loc);
```

Description

mblen_l determines the number of bytes contained in the multi-byte character pointed to by **s** in the locale **loc**.

If **s** is a null pointer, **mblen_l** returns a nonzero or zero value, if multi-byte character encodings, respectively, do or do not have state-dependent encodings

If s is not a null pointer, **mblen_l** either returns 0 (if **s** points to the null character), or returns the number of bytes that are contained in the multi-byte character (if the next **n** or fewer bytes form a valid multi-byte character), or returns 1 (if they do not form a valid multi-byte character).

Note

Except that the conversion state of the **mbtowc_l** function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, s, n, loc);
```

Note

It is guaranteed that no library function in the Standard C library calls **mblen_l**.

See Also

mblen_l, **mbtowc_l**

mbstowcs

Synopsis

```
size_t mbstowcs(wchar_t *pwcs,
                const char *s,
                size_t n);
```

Description

mbstowcs 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 not more than **n** wide characters into the array pointed to by **pwcs**.

No multi-byte characters that follow a null character (which is converted into a null wide character) will be examined or converted. Each multi-byte character is converted as if by a call to the **mbtowc** function, except that the conversion state of the **mbtowc** function is not affected.

No more than **n** elements will be modified in the array pointed to by **pwcs**. If copying takes place between objects that overlap, the behavior is undefined.

mbstowcs returns 1 if an invalid multi-byte character is encountered, otherwise **mbstowcs** returns the number of array elements modified (if any), not including a terminating null wide character.

mbstowcs_l

Synopsis

```
size_t mbstowcs_l(wchar_t *pwcs,
                  const char *s,
                  size_t n,
                  __locale_s *loc);
```

Description

mbstowcs_l is as **mbstowcs** except that the local **loc** is used for the conversion as opposed to the current locale.

See Also

mbstowcs.

mbtowc

Synopsis

```
int mbtowc(wchar_t *pwc,
           const char *s,
           size_t n);
```

Description

mbtowc converts a single multi-byte character to a wide character in the current locale.

If **s** is a null pointer, **mbtowc** returns a nonzero value if multi-byte character encodings are state-dependent in the current locale, and zero otherwise.

If **s** is not null and the object that **s** points to is a wide-character null character, **mbtowc** returns 0.

If **s** is not null and the object that points to forms a valid multi-byte character, **mbtowc** returns the length in bytes of the multi-byte character.

If the object that points to does not form a valid multi-byte character within the first **n** characters, it returns 1.

See Also

mbtowc_l

mbtowc_l

Synopsis

```
int mbtowc_l(wchar_t *pwc,
             const char *s,
             size_t n,
             __locale_s *loc);
```

Description

mbtowc_l converts a single multi-byte character to a wide character in locale **loc**.

If **s** is a null pointer, **mbtowc_l** returns a nonzero value if multi-byte character encodings are state-dependent in the locale **loc**, and zero otherwise.

If **s** is not null and the object that **s** points to is a wide-character null character, **mbtowc_l** returns 0.

If **s** is not null and the object that points to forms a valid multi-byte character, **mbtowc_l** returns the length in bytes of the multi-byte character.

If the object that **s** points to does not form a valid multi-byte character within the first **n** characters, it returns 1.

See Also

mbtowc

qsort

Synopsis

```
void qsort(void *buf,
           size_t num,
           size_t size,
           int (*compare)(const void *, const void *));
```

qsort sorts the array ***base** using the **compare** function. The array should have **num** elements of **size** bytes. The **compare** function should return a negative value if the first parameter is less than second parameter, zero if the parameters are equal and a positive value if the first parameter is greater than the second parameter.

rand

Synopsis

```
int rand(void);
```

Description

rand computes a sequence of pseudo-random integers in the range 0 to **RAND_MAX**.

rand returns the computed pseudo-random integer.

realloc

Synopsis

```
void *realloc(void *p,
              size_t size);
```

Description

realloc deallocates the old object pointed to by **ptr** and returns a pointer to a new object that has the size specified by **size**. The contents of the new object is identical to that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

If **ptr** is a null pointer, **realloc** behaves like realloc for the specified size. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

realloc returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object could not be allocated.

If **ptr** does not match a pointer earlier returned by **calloc**, **malloc**, or **realloc**, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

srand

Synopsis

```
void srand(unsigned int seed);
```

Description

srand uses the argument **seed** as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**. If **srand** is called with the same seed value, the same sequence of pseudo-random numbers is generated.

If **rand** is called before any calls to **srand** have been made, a sequence is generated as if **srand** is first called with a seed value of 1.

See Also

rand

strtod

Synopsis

```
double strtod(const char *nptr,
              char **endptr);
```

Description

strtod converts the initial portion of the string pointed to by **nptr** to a **double** representation.

First, **strtod** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling a floating-point constant, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtod** then attempts to convert the subject sequence to a floating-point number, and return the result.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **strtod**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtod returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **HUGE_VAL** is returned according to the sign of the value, if any, and the value of the macro **errno** is stored in **errno**.

strtof

Synopsis

```
float strtof(const char *nptr,
             char **endptr);
```

Description

strtof converts the initial portion of the string pointed to by **nptr** to a **double** representation.

First, **strtof** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling a floating-point constant, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtof** then attempts to convert the subject sequence to a floating-point number, and return the result.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtof returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **HUGE_VALF** is returned according to the sign of the value, if any, and the value of the macro **errno** is stored in **errno**.

strtol

Synopsis

```
long int strtol(const char *nptr,
                char **endptr,
                int base);
```

Description

strtol converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtol** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtol** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtol returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MIN** or **LONG_MAX** is returned according to the sign of the value, if any, and the value of the macro **errno** is stored in **errno**.

strtoll

Synopsis

```
long long int strtoll(const char *nptr,
                      char **endptr,
                      int base);
```

Description

strtoll converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoll** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoll** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtoll returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LLONG_MIN** or **LLONG_MAX** is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in **errno**.

strtoul

Synopsis

```
unsigned long int strtoul(const char *nptr,
                          char **endptr,
                          int base);
```

Description

strtoul converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoul** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoul** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtoul returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MAX** or **ULONG_MAX** is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in **errno**.

strtoull

Synopsis

```
unsigned long long int strtoull(const char *nptr,
                                char **endptr,
                                int base);
```

Description

strtoull converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoull** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoull** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtoull returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LLONG_MAX** or **ULLONG_MAX** is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in **errno**.

ulltoa

Synopsis

```
char *ulltoa(unsigned long long val,
             char *buf,
             int radix);
```

Description

ulltoa converts **val** to a string in base **radix** and places the result in **buf**.

ulltoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

See Also

itoa, **ltoa**, **lltoa**, **ultoa**, **utoa**

ultoa

Synopsis

```
char *ultoa(unsigned long val,
            char *buf,
            int radix);
```

Description

ultoa converts **val** to a string in base **radix** and places the result in **buf**.

ultoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

See Also

itoa, ltoa, lltoa, ulltoa, utoa

utoa

Synopsis

```
char *utoa(unsigned val,
           char *buf,
           int radix);
```

Description

utoa converts **val** to a string in base **radix** and places the result in **buf**.

utoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

See Also

itoa, **ltoa**, **lltoa**, **ultoa**, **ulltoa**

<string.h>

Overview

The header file <string.h> defines functions that operate on arrays that are interpreted as null-terminated strings.

Various methods are used for determining the lengths of the arrays, but in all cases a **char** * or **void** * argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

Where an argument declared as **size_t** *n* specifies the length of an array for a function, *n* can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function, pointer arguments must have valid values on a call with a zero size. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

API Summary

Copying functions	
memccpy	Copy memory with specified terminator (POSIX extension)
memcpy	Copy memory
memcpy_fast	Copy memory
memmove	Safely copy overlapping memory
mempcpy	Copy memory (GNU extension)
strcat	Concatenate strings
strcpy	Copy string
strdup	Duplicate string (POSIX extension)
strlcat	Copy string up to a maximum length with terminator (BSD extension)
strlcpy	Copy string up to a maximum length with terminator (BSD extension)
strncat	Concatenate strings up to maximum length
strncpy	Copy string up to a maximum length
strndup	Duplicate string (POSIX extension)
Comparison functions	
memcmp	Compare memory
strcasecmp	Compare strings ignoring case (POSIX extension)

strcmp	Compare strings
strncasecmp	Compare strings up to a maximum length ignoring case (POSIX extension)
strncmp	Compare strings up to a maximum length
Search functions	
memchr	Search memory for a character
strcasestr	Find first case-insensitive occurrence of a string within string
strchr	Find character within string
strcspn	Compute size of string not prefixed by a set of characters
strncasestr	Find first case-insensitive occurrence of a string within length-limited string
strnchr	Find character in a length-limited string
strnlen	Calculate length of length-limited string (POSIX extension)
strnstr	Find first occurrence of a string within length-limited string
strpbrk	Find first occurrence of characters within string
strrchr	Find last occurrence of character within string
strsep	Break string into tokens (4.4BSD extension)
strspn	Compute size of string prefixed by a set of characters
strstr	Find first occurrence of a string within string
strtok	Break string into tokens
strtok_r	Break string into tokens, reentrant version (POSIX extension)
Miscellaneous functions	
memset	Set memory to character
strerror	Decode error code
strlen	Calculate length of string

memccpy

Synopsis

```
void *memccpy(void *s1,
              const void *s2,
              int c,
              size_t n);
```

Description

memccpy copies at most **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The copying stops as soon as **n** characters are copied or the character **c** is copied into the destination object pointed to by **s1**. The behavior of **memccpy** is undefined if copying takes place between objects that overlap.

memccpy returns a pointer to the character immediately following **c** in **s1**, or **NULL** if **c** was not found in the first **n** characters of **s2**.

Note

memccpy conforms to POSIX.1-2008.

memchr

Synopsis

```
void *memchr(const void *s,
             int c,
             size_t n);
```

Description

memchr locates the first occurrence of **c** (converted to an **unsigned char**) in the initial **n** characters (each interpreted as **unsigned char**) of the object pointed to by **s**. Unlike **strchr**, **memchr** does *not* terminate a search when a null character is found in the object pointed to by **s**.

memchr returns a pointer to the located character, or a null pointer if **c** does not occur in the object.

memcmp

Synopsis

```
int memcmp(const void *s1,
           const void *s2,
           size_t n);
```

Description

memcmp compares the first **n** characters of the object pointed to by **s1** to the first **n** characters of the object pointed to by **s2**. **memcmp** returns an integer greater than, equal to, or less than zero as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

memcpy

Synopsis

```
void *memcpy(void *s1,
             const void *s2,
             size_t n);
```

Description

memcpy copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **memcpy** is undefined if copying takes place between objects that overlap.

memcpy returns the value of **s1**.

memcpy_fast

Synopsis

```
void *memcpy_fast(void *s1,
                  const void *s2,
                  size_t n);
```

Description

memcpy_fast copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **memcpy_fast** is undefined if copying takes place between objects that overlap. The implementation of **memcpy_fast** is optimized for speed for all cases of memcpy and as such has a large code memory requirement. This function is implemented for little-endian ARM and 32-bit Thumb-2 instruction sets only.

memcpy_fast returns the value of **s1**.

memmove

Synopsis

```
void *memmove(void *s1,
              const void *s2,
              size_t n);
```

Description

memmove copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1** ensuring that if **s1** and **s2** overlap, the copy works correctly. Copying takes place as if the **n** characters from the object pointed to by **s2** are first copied into a temporary array of **n** characters that does not overlap the objects pointed to by **s1** and **s2**, and then the **n** characters from the temporary array are copied into the object pointed to by **s1**.

memmove returns the value of **s1**.

mempcpy

Synopsis

```
void *mempcpy(void *s1,
              const void *s2,
              size_t n);
```

Description

mempcpy copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **mempcpy** is undefined if copying takes place between objects that overlap.

mempcpy returns a pointer to the byte following the last written byte.

Note

This is an extension found in GNU libc.

memset

Synopsis

```
void *memset(void *s,
             int c,
             size_t n);
```

Description

memset copies the value of **c** (converted to an **unsigned char**) into each of the first **n** characters of the object pointed to by **s**.

memset returns the value of **s**.

strcasecmp

Synopsis

```
int strcasecmp(const char *s1,
               const char *s2);
```

Description

strcasecmp compares the string pointed to by **s1** to the string pointed to by **s2** ignoring differences in case.
strcasecmp returns an integer greater than, equal to, or less than zero if the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

Note

strcasecmp conforms to POSIX.1-2008.

strcasestr

Synopsis

```
char *strcasestr(const char *s1,
                 const char *s2);
```

Description

strcasestr locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2** without regard to character case.

strcasestr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **strcasestr** returns **s1**.

Note

strcasestr is an extension commonly found in Linux and BSD C libraries.

strcat

Synopsis

```
char *strcat(char *s1,
             const char *s2);
```

Description

strcat appends a copy of the string pointed to by **s2** (including the terminating null character) to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. The behavior of **strcat** is undefined if copying takes place between objects that overlap.

strcat returns the value of **s1**.

strchr

Synopsis

```
char *strchr(const char *s,
             int c);
```

Description

strchr locates the first occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

strchr returns a pointer to the located character, or a null pointer if **c** does not occur in the string.

strcmp

Synopsis

```
int strcmp(const char *s1,
           const char *s2);
```

Description

strcmp compares the string pointed to by **s1** to the string pointed to by **s2**. **strcmp** returns an integer greater than, equal to, or less than zero if the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

strcpy

Synopsis

```
char *strcpy(char *s1,
             const char *s2);
```

Description

strcpy copies the string pointed to by **s2** (including the terminating null character) into the array pointed to by **s1**. The behavior of **strcpy** is undefined if copying takes place between objects that overlap.

strcpy returns the value of **s1**.

strcspn

Synopsis

```
size_t strcspn(const char *s1,
               const char *s2);
```

Description

strcspn computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters not from the string pointed to by **s2**.

strcspn returns the length of the segment.

strdup

Synopsis

```
char *strdup(const char *s1);
```

Description

strdup duplicates the string pointed to by **s1** by using **malloc** to allocate memory for a copy of **s** and then copying **s**, including the terminating null, to that memory **strdup** returns a pointer to the new string or a null pointer if the new string cannot be created. The returned pointer can be passed to **free**.

Note

strdup conforms to POSIX.1-2008 and SC22 TR 24731-2.

strerror

Synopsis

```
char *strerror(int num);
```

Description

strerror maps the number in **num** to a message string. Typically, the values for **num** come from **errno**, but **strerror** can map any value of type **int** to a message.

strerror returns a pointer to the message string. The program must not modify the returned message string. The message may be overwritten by a subsequent call to **strerror**.

strlcat

Synopsis

```
size_t strlcat(char *s1,
               const char *s2,
               size_t n);
```

Description

strlcat appends no more than **nstrlen**(**dst**)1 characters pointed to by **s2** into the array pointed to by **s1** and always terminates the result with a null character if **n** is greater than zero. Both the strings **s1** and **s2** must be terminated with a null character on entry to **strlcat** and a byte for the terminating null should be included in **n**. The behavior of **strlcat** is undefined if copying takes place between objects that overlap.

strlcat returns the number of characters it tried to copy, which is the sum of the lengths of the strings **s1** and **s2** or **n**, whichever is smaller.

Note

strlcat is commonly found in OpenBSD libraries.

strlcpy

Synopsis

```
size_t strlcpy(char *s1,
               const char *s2,
               size_t n);
```

Description

strlcpy copies up to **n**1 characters from the string pointed to by **s2** into the array pointed to by **s1** and always terminates the result with a null character. The behavior of **strlcpy** is undefined if copying takes place between objects that overlap.

strlcpy returns the number of characters it tried to copy, which is the length of the string **s2** or **n**, whichever is smaller.

Note

strlcpy is commonly found in OpenBSD libraries and contrasts with **strncpy** in that the resulting string is always terminated with a null character.

strlen

Synopsis

```
size_t strlen(const char *s);
```

Description

strlen returns the length of the string pointed to by **s**, that is the number of characters that precede the terminating null character.

strncasecmp

Synopsis

```
int strncasecmp(const char *s1,
                const char *s2,
                size_t n);
```

Description

strncasecmp compares not more than **n** characters from the array pointed to by **s1** to the array pointed to by **s2** ignoring differences in case. Characters that follow a null character are not compared.

strncasecmp returns an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

Note

strncasecmp conforms to POSIX.1-2008.

strncasestr

Synopsis

```
char *strncasestr(const char *s1,
                  const char *s2,
                  size_t n);
```

Description

strncasestr searches at most **n** characters to locate the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2** without regard to character case.

strncasestr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **strncasestr** returns **s1**.

Note

strncasestr is an extension commonly found in Linux and BSD C libraries.

strncat

Synopsis

```
char *strncat(char *s1,
              const char *s2,
              size_t n);
```

Description

strncat appends not more than **n** characters from the array pointed to by **s2** to the end of the string pointed to by **s1**. A null character in **s1** and characters that follow it are not appended. The initial character of **s2** overwrites the null character at the end of **s1**. A terminating null character is always appended to the result. The behavior of **strncat** is undefined if copying takes place between objects that overlap.

strncat returns the value of **s1**.

strnchr

Synopsis

```
char *strnchr(const char *str,
              size_t n,
              int ch);
```

Description

strnchr searches not more than **n** characters to locate the first occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

strnchr returns a pointer to the located character, or a null pointer if **c** does not occur in the string.

strncmp

Synopsis

```
int strncmp(const char *s1,
            const char *s2,
            size_t n);
```

Description

strncmp compares not more than **n** characters from the array pointed to by **s1** to the array pointed to by **s2**. Characters that follow a null character are not compared.

strncmp returns an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

strncpy

Synopsis

```
char *strncpy(char *s1,
              const char *s2,
              size_t n);
```

Description

strncpy copies not more than **n** characters from the array pointed to by **s2** to the array pointed to by **s1**. Characters that follow a null character in **s2** are not copied. The behavior of **strncpy** is undefined if copying takes place between objects that overlap. If the array pointed to by **s2** is a string that is shorter than **n** characters, null characters are appended to the copy in the array pointed to by **s1**, until **n** characters in all have been written.

strncpy returns the value of **s1**.

Note

No null character is implicitly appended to the end of **s1**, so **s1** will only be terminated by a null character if the length of the string pointed to by **s2** is less than **n**.

strndup

Synopsis

```
char *strndup(const char *s1,
              size_t n);
```

Description

strndup duplicates at most **n** characters from the the string pointed to by **s1** by using **malloc** to allocate memory for a copy of **s1**.

If the length of string pointed to by **s1** is greater than **n** characters, only **n** characters will be duplicated. If **n** is greater than the length of string pointed to by **s1**, all characters in the string are copied into the allocated array including the terminating null character.

strndup returns a pointer to the new string or a null pointer if the new string cannot be created. The returned pointer can be passed to **free**.

Note

strndup conforms to POSIX.1-2008 and SC22 TR 24731-2.

strnlen

Synopsis

```
size_t strnlen(const char *s,
               size_t n);
```

Description

strnlen returns the length of the string pointed to by **s**, up to a maximum of **n** characters. **strnlen** only examines the first **n** characters of the string **s**.

Note

strnlen conforms to POSIX.1-2008.

strnstr

Synopsis

```
char *strnstr(const char *s1,
              const char *s2,
              size_t n);
```

Description

strnstr searches at most **n** characters to locate the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**.

strnstr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **strnstr** returns **s1**.

Note

strnstr is an extension commonly found in Linux and BSD C libraries.

strpbrk

Synopsis

```
char *strpbrk(const char *s1,
              const char *s2);
```

Description

strpbrk locates the first occurrence in the string pointed to by **s1** of any character from the string pointed to by **s2**.

strpbrk returns a pointer to the character, or a null pointer if no character from **s2** occurs in **s1**.

strrchr

Synopsis

```
char *strrchr(const char *s,
              int c);
```

Description

strrchr locates the last occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

strrchr returns a pointer to the character, or a null pointer if **c** does not occur in the string.

strsep

Synopsis

```
char *strsep(char **stringp,
             const char *delim);
```

Description

strsep locates, in the string referenced by ***stringp**, the first occurrence of any character in the string **delim** (or the terminating null character) and replaces it with a null character. The location of the next character after the delimiter character (or NULL, if the end of the string was reached) is stored in ***stringp**. The original value of ***stringp** is returned.

An empty field (that is, a character in the string **delim** occurs as the first character of ***stringp** can be detected by comparing the location referenced by the returned pointer to the null character.

If ***stringp** is initially null, **strsep** returns null.

Note

strsep is an extension commonly found in Linux and BSD C libraries.

strspn

Synopsis

```
size_t strspn(const char *s1,
              const char *s2);
```

Description

strspn computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

strspn returns the length of the segment.

strstr

Synopsis

```
char *strstr(const char *s1,
             const char *s2);
```

Description

strstr locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**.

strstr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **strstr** returns **s1**.

strtok

Synopsis

```
char *strtok(char *s1,
             const char *s2);
```

Description

strtok A sequence of calls to **strtok** breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator string pointed to by **s2** may be different from call to call.

The first call in the sequence searches the string pointed to by **s1** for the first character that is not contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and **strtok** returns a null pointer. If such a character is found, it is the start of the first token.

strtok then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. **strtok** saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Note

strtok maintains static state and is therefore not reentrant and not thread safe. See **strtok_r** for a thread-safe and reentrant variant.

See Also

strsep, **strtok_r**.

strtok_r

Synopsis

```
char *strtok_r(char *s1,
               const char *s2,
               char **s3);
```

Description

strtok_r is a reentrant version of the function **strtok** where the state is maintained in the object of type **char *** pointed to by **s3**.

Note

strtok_r conforms to POSIX.1-2008 and is commonly found in Linux and BSD C libraries.

See Also

strtok.

<time.h>

API Summary

Types	
clock_t	Clock type
time_t	Time type
tm	Time structure
Functions	
asctime	Convert a struct tm to a string
asctime_r	Convert a struct tm to a string
ctime	Convert a time_t to a string
ctime_r	Convert a time_t to a string
difftime	Calculates the difference between two times
gmtime	Convert a time_t to a struct tm
gmtime_r	Convert a time_t to a struct tm
localtime	Convert a time_t to a struct tm
localtime_r	Convert a time_t to a struct tm
mktime	Convert a struct tm to time_t
strftime	Format a struct tm to a string

asctime

Synopsis

```
char *asctime(const tm *tp);
```

Description

asctime converts the ***tp** struct to a null terminated string of the form `Sun Sep 16 01:03:52 1973`. The returned string is held in a static buffer. **asctime** is not re-entrant.

asctime_r

Synopsis

```
char *asctime_r(const tm *tp,
                char *buf);
```

Description

asctime_r converts the ***tp** struct to a null terminated string of the form Sun Sep 16 01:03:52 1973 in **buf** and returns **buf**. The **buf** must point to an array at least 26 bytes in length.

clock_t

Synopsis

```
typedef long clock_t;
```

Description

clock_t is the type returned by the **clock** function.

ctime

Synopsis

```
char *ctime(const time_t *tp);
```

Description

ctime converts the ***tp** to a null terminated string. The returned string is held in a static buffer, this function is not re-entrant.

ctime_r

Synopsis

```
char *ctime_r(const time_t *tp,
              char *buf);
```

Description

ctime_r converts the ***tp** to a null terminated string in **buf** and returns **buf**. The **buf** must point to an array at least 26 bytes in length.

difftime

Synopsis

```
double difftime(time_t time2,
                time_t time1);
```

Description

difftime returns **time1** - **time0** as a double precision number.

gmtime

Synopsis

```
gmtime(const time_t *tp);
```

Description

gmtime converts the ***tp** time format to a **struct** tm time format. The returned value points to a static object - this function is not re-entrant.

gmtime_r

Synopsis

```
gmtime_r(const time_t *tp,
         tm *result);
```

Description

gmtime_r converts the ***tp** time format to a **struct** tm time format in ***result** and returns **result**.

localtime

Synopsis

```
localtime(const time_t *tp);
```

Description

localtime converts the ***tp** time format to a **struct** tm local time format. The returned value points to a static object - this function is not re-entrant.

localtime_r

Synopsis

```
localtime_r(const time_t *tp,
            tm *result);
```

Description

localtime_r converts the ***tp** time format to a **struct** tm local time format in ***result** and returns **result**.

mktime

Synopsis

```
time_t mktime(tm *tp);
```

Description

mktime validates (and updates) the ***tp** struct to ensure that the **tm_sec**, **tm_min**, **tm_hour**, **tm_mon** fields are within the supported integer ranges and the **tm_mday**, **tm_mon** and **tm_year** fields are consistent. The validated ***tp** struct is converted to the number of seconds since UTC 1 January 1970 and returned.

strftime

Synopsis

```
size_t strftime(char *s,
                size_t smax,
                const char *fmt,
                const tm *tp);
```

Description

strftime formats the ***tp** struct to a null terminated string of maximum size **smax**-1 into the array at ***s** based on the **fmt** format string. The format string consists of conversion specifications and ordinary characters. Conversion specifications start with a **%** character followed by an optional **#** character. The following conversion specifications are supported:

Specification	Description
%s	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for locale
%#c	Date and time formatted as "%A, %B %#d, %Y, %H:%M:%S" (Microsoft extension)
%C	Century number
%d	Day of month as a decimal number [01,31]
%#d	Day of month without leading zero [1,31]
%D	Date in the form %m/%d/%y (POSIX.1-2008 extension)
%e	Day of month [1,31], single digit preceded by space
%F	Date in the format %Y-%m-%d
%h	Abbreviated month name as %b
%H	Hour in 24-hour format [00,23]
%#H	Hour in 24-hour format without leading zeros [0,23]
%I	Hour in 12-hour format [01,12]
%#I	Hour in 12-hour format without leading zeros [1,12]
%j	Day of year as a decimal number [001,366]
%#j	Day of year as a decimal number without leading zeros [1,366]
%k	Hour in 24-hour clock format [0,23] (POSIX.1-2008 extension)

%I	Hour in 12-hour clock format [0,12] (POSIX.1-2008 extension)
%m	Month as a decimal number [01,12]
%#m	Month as a decimal number without leading zeros [1,12]
%M	Minute as a decimal number [00,59]
%#M	Minute as a decimal number without leading zeros [0,59]
%n	Insert newline character (POSIX.1-2008 extension)
%p	Locale's a.m or p.m indicator for 12-hour clock
%r	Time as %I:%M:%s %p (POSIX.1-2008 extension)
%R	Time as %H:%M (POSIX.1-2008 extension)
%S	Second as a decimal number [00,59]
%t	Insert tab character (POSIX.1-2008 extension)
%T	Time as %H:%M:%S
%#S	Second as a decimal number without leading zeros [0,59]
%U	Week of year as a decimal number [00,53], Sunday is first day of the week
%#U	Week of year as a decimal number without leading zeros [0,53], Sunday is first day of the week
%w	Weekday as a decimal number [0,6], Sunday is 0
%W	Week number as a decimal number [00,53], Monday is first day of the week
%#W	Week number as a decimal number without leading zeros [0,53], Monday is first day of the week
%x	Locale's date representation
%#x	Locale's long date representation
%X	Locale's time representation
%y	Year without century, as a decimal number [00,99]
%#y	Year without century, as a decimal number without leading zeros [0,99]
%Y	Year with century, as decimal number
%z,%Z	Timezone name or abbreviation
%%	%

time_t

Synopsis

```
typedef long time_t;
```

Description

time_t is a long type that represents the time in number of seconds since UTC 1 January 1970, negative values indicate time before UTC 1 January 1970.

tm

Synopsis

```
typedef struct {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
} tm;
```

Description

tm structure has the following fields.

Member	Description
tm_sec	seconds after the minute - [0,59]
tm_min	minutes after the hour - [0,59]
tm_hour	hours since midnight - [0,23]
tm_mday	day of the month - [1,31]
tm_mon	months since January - [0,11]
tm_year	years since 1900
tm_wday	days since Sunday - [0,6]
tm_yday	days since January 1 - [0,365]
tm_isdst	daylight savings time flag

<wchar.h>

API Summary

Character minimum and maximum values	
WCHAR_MAX	Maximum value of a wide character
WCHAR_MIN	Minimum value of a wide character
Constants	
WEOF	End of file indication
Types	
wchar_t	Wide character type
wint_t	Wide integer type
Copying functions	
wcscat	Concatenate strings
wcscpy	Copy string
wcsncat	Concatenate strings up to maximum length
wcsncpy	Copy string up to a maximum length
wmemccpy	Copy memory with specified terminator (POSIX extension)
wmemcpy	Copy memory
wmemmove	Safely copy overlapping memory
wmempcpy	Copy memory (GNU extension)
Comparison functions	
wcscmp	Compare strings
wcsncmp	Compare strings up to a maximum length
wmemcmp	Compare memory
Search functions	
wcschr	Find character within string
wcscspn	Compute size of string not prefixed by a set of characters
wcsnchr	Find character in a length-limited string
wcsnlen	Calculate length of length-limited string
wcsnstr	Find first occurrence of a string within length-limited string
wcspbrk	Find first occurrence of characters within string
wcsrchr	Find last occurrence of character within string

wcsspn	Compute size of string prefixed by a set of characters
wcsstr	Find first occurrence of a string within string
wcstok	Break string into tokens
wcstok_r	Break string into tokens (reentrant version)
wmemchr	Search memory for a wide character
wstrsep	Break string into tokens
Miscellaneous functions	
wcsdup	Duplicate string
wcslen	Calculate length of string
wmemset	Set memory to wide character
Multi-byte/wide string conversion functions	
mbrtowc	Convert multi-byte character to wide character
mbrtowc_l	Convert multi-byte character to wide character
msbinit	Query conversion state
wcrtomb	Convert wide character to multi-byte character (restartable)
wcrtomb_l	Convert wide character to multi-byte character (restartable)
wctob	Convert wide character to single-byte character
wctob_l	Convert wide character to single-byte character
Multi-byte to wide character conversions	
mbrlen	Determine number of bytes in a multi-byte character
mbrlen_l	Determine number of bytes in a multi-byte character
mbsrtowcs	Convert multi-byte string to wide character string
mbsrtowcs_l	Convert multi-byte string to wide character string
Single-byte to wide character conversions	
btowc	Convert single-byte character to wide character
btowc_l	Convert single-byte character to wide character

WCHAR_MAX

Synopsis

```
#define WCHAR_MAX  ...
```

Description

WCHAR_MAX is the maximum value for an object of type **wchar_t**. Although capable of storing larger values, the maximum value implemented by the conversion functions in the library is the value 0x10FFFF defined by ISO 10646.

WCHAR_MIN

Synopsis

```
#define WCHAR_MIN  ...
```

Description

WCHAR_MIN is the minimum value for an object of type **wchar_t**.

WEOF

Synopsis

```
#define WEOF ((wint_t)~0U)
```

Description

WEOF expands to a constant value that does not correspond to any character in the wide character set. It is typically used to indicate an end of file condition.

btowc

Synopsis

```
wint_t btowc(int c);
```

Description

btowc function determines whether **c** constitutes a valid single-byte character. If **c** is a valid single-byte character, **btowc** returns the wide character representation of that character

btowc 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.

btowc_l

Synopsis

```
wint_t btowc_l(int c,
               locale_t loc);
```

Description

btowc_l function determines whether **c** constitutes a valid single-byte character in the locale **loc**. If **c** is a valid single-byte character, **btowc_l** returns the wide character representation of that character

btowc_l 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.

mbrlen

Synopsis

```
size_t mbrlen(const char *s,
              size_t n,
              mbstate_t *ps);
```

Note

mbrlen function 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.

mbrlen_l

Synopsis

```
size_t mbrlen_l(const char *s,
                size_t n,
                mbstate_t *ps,
                locale_t loc);
```

Note

mbrlen_l function is equivalent to the call:

```
mbrtowc_l(NULL, s, n, ps != NULL ? ps : &internal, loc);
```

where **internal** is the **mbstate_t** object for the **mbrlen** function, except that the expression designated by **ps** is evaluated only once.

mbrtowc

Synopsis

```
size_t mbrtowc(wchar_t *pwc,
               const char *s,
               size_t n,
               mbstate_t *ps);
```

Description

mbrtowc converts a single multi-byte character to a wide character in the current locale.

If **s** is a null pointer, **mbrtowc** is equivalent to `mbrtowc(NULL, "", 1, ps)`, ignoring **pwc** and **n**.

If **s** is not null and the object that **s** points to is a wide-character null character, **mbrtowc** returns 0.

If **s** is not null and the object that points to forms a valid multi-byte character with a most **n** bytes, **mbrtowc** returns the length in bytes of the multi-byte character and stores that wide character to the object pointed to by **pwc** (if **pwc** is not null).

If the object that points to forms an incomplete, but possibly valid, multi-byte character, **mbrtowc** returns 2. If the object that points to does not form a partial multi-byte character, **mbrtowc** returns 1.

See Also

mbtowc, **mbrtowc_l**

mbrtowc_l

Synopsis

```
size_t mbrtowc_l(wchar_t *pwc,
                 const char *s,
                 size_t n,
                 mbstate_t *ps,
                 locale_t loc);
```

Description

mbrtowc_l converts a single multi-byte character to a wide character in the locale **loc**.

If **s** is a null pointer, **mbrtowc_l** is equivalent to `mbrtowc(NULL, "", 1, ps)`, ignoring **pwc** and **n**.

If **s** is not null and the object that **s** points to is a wide-character null character, **mbrtowc_l** returns 0.

If **s** is not null and the object that points to forms a valid multi-byte character with a most **n** bytes, **mbrtowc_l** returns the length in bytes of the multi-byte character and stores that wide character to the object pointed to by **pwc** (if **pwc** is not null).

If the object that points to forms an incomplete, but possibly valid, multi-byte character, **mbrtowc_l** returns 2. If the object that points to does not form a partial multi-byte character, **mbrtowc_l** returns 1.

See Also

mbrtowc, mbtowc_l

mbsrtowcs

Synopsis

```
size_t mbsrtowcs(wchar_t *dst,
                 const char **src,
                 size_t len,
                 mbstate_t *ps);
```

Description

mbsrtowcs 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 If **dst** is not a null pointer, the converted characters are stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null character, which is also stored.

Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multi-byte character, or (if **dst** is not a null pointer) when **len** wide characters have been stored into the array pointed to by **dst**. Each conversion takes place as if by a call to the **mbrtowc** function.

If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multi-byte character converted (if any). If conversion stopped due to reaching a terminating null character and if **dst** is not a null pointer, the resulting state described is the initial conversion state.

See Also

mbsrtowcs_l, mbrtowc

mbsrtowcs_l

Synopsis

```
size_t mbsrtowcs_l(wchar_t *dst,
                   const char **src,
                   size_t len,
                   mbstate_t *ps,
                   locale_t loc);
```

Description

mbsrtowcs_l 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 If **dst** is not a null pointer, the converted characters are stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null character, which is also stored.

Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multi-byte character, or (if **dst** is not a null pointer) when **len** wide characters have been stored into the array pointed to by **dst**. Each conversion takes place as if by a call to the **mbrtowc** function.

If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multi-byte character converted (if any). If conversion stopped due to reaching a terminating null character and if **dst** is not a null pointer, the resulting state described is the initial conversion state.

See Also

mbsrtowcs_l, mbrtowc

msbinit

Synopsis

```
int msbinit(const mbstate_t *ps);
```

Description

msbinit function returns nonzero if **ps** is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, **msbinit** returns zero.

wchar_t

Synopsis

```
typedef __RAL_WCHAR_T wchar_t;
```

Description

wchar_t holds a single wide character.

Depending on implementation you can control whether **wchar_t** is represented by a short 16-bit type or the standard 32-bit type.

wcrtomb

Synopsis

```
size_t wcrtomb(char *s,
               wchar_t wc,
               mbstate_t *ps);
```

If **s** is a null pointer, **wcrtomb** function is equivalent to the call `wcrtomb(buf, L'\0', ps)` where **buf** is an internal buffer.

If **s** is not a null pointer, **wcrtomb** determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **wc**, and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **wc** is a null wide character, a null byte is stored; the resulting state described is the initial conversion state.

wcrtomb returns the number of bytes stored in the array object. When **wc** is not a valid wide character, an encoding error occurs: **wcrtomb** stores the value of the macro **EILSEQ** in **errno** and returns `(size_t)(-1)`; the conversion state is unspecified.

wcrtomb_l

Synopsis

```
size_t wcrtomb_l(char *s,
                 wchar_t wc,
                 mbstate_t *ps,
                 locale_t loc);
```

If **s** is a null pointer, **wcrtomb_l** function is equivalent to the call `wcrtomb_l(buf, L'\0', ps, loc)` where **buf** is an internal buffer.

If **s** is not a null pointer, **wcrtomb_l** determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **wc**, and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **wc** is a null wide character, a null byte is stored; the resulting state described is the initial conversion state.

wcrtomb_l returns the number of bytes stored in the array object. When **wc** is not a valid wide character, an encoding error occurs: **wcrtomb_l** stores the value of the macro **EILSEQ** in **errno** and returns `(size_t)(-1)`; the conversion state is unspecified.

wcscat

Synopsis

```
wchar_t *wcscat(wchar_t *s1,
                const wchar_t *s2);
```

Description

wcscat appends a copy of the wide string pointed to by **s2** (including the terminating null wide character) to the end of the wide string pointed to by **s1**. The initial character of **s2** overwrites the null wide character at the end of **s1**. The behavior of **wcscat** is undefined if copying takes place between objects that overlap.

wcscat returns the value of **s1**.

wcschr

Synopsis

```
wchar_t *wcschr(const wchar_t *s,
                wchar_t c);
```

Description

wcschr locates the first occurrence of **c** in the wide string pointed to by **s**. The terminating wide null character is considered to be part of the string.

wcschr returns a pointer to the located wide character, or a null pointer if **c** does not occur in the string.

wcscmp

Synopsis

```
int wcscmp(const wchar_t *s1,
           const wchar_t *s2);
```

Description

wcscmp compares the wide string pointed to by **s1** to the wide string pointed to by **s2**. **wcscmp** returns an integer greater than, equal to, or less than zero if the wide string pointed to by **s1** is greater than, equal to, or less than the wide string pointed to by **s2**.

wcscpy

Synopsis

```
wchar_t *wcscpy(wchar_t *s1,
                const wchar_t *s2);
```

Description

wcscpy copies the wide string pointed to by **s2** (including the terminating null wide character) into the array pointed to by **s1**. The behavior of **wcscpy** is undefined if copying takes place between objects that overlap.

wcscpy returns the value of **s1**.

wcscspn

Synopsis

```
size_t wcscspn(const wchar_t *s1,
               const wchar_t *s2);
```

Description

wcscspn computes the length of the maximum initial segment of the wide string pointed to by **s1** which consists entirely of wide characters not from the wide string pointed to by **s2**.

wcscspn returns the length of the segment.

wcsdup

Synopsis

```
wchar_t *wcsdup(const wchar_t *s1);
```

Description

wcsdup duplicates the wide string pointed to by **s1** by using **malloc** to allocate memory for a copy of **s** and then copying **s**, including the terminating wide null character, to that memory. The returned pointer can be passed to **free**. **wcsdup** returns a pointer to the new wide string or a null pointer if the new string cannot be created.

Note

wcsdup is an extension commonly found in Linux and BSD C libraries.

wcslen

Synopsis

```
size_t wcslen(const wchar_t *s);
```

Description

wcslen returns the length of the wide string pointed to by **s**, that is the number of wide characters that precede the terminating null wide character.

wcsncat

Synopsis

```
wchar_t *wcsncat(wchar_t *s1,
                 const wchar_t *s2,
                 size_t n);
```

Description

wcsncat appends not more than **n** wude characters from the array pointed to by **s2** to the end of the wide string pointed to by **s1**. A null wide character in **s1** and wide characters that follow it are not appended. The initial wide character of **s2** overwrites the null wide character at the end of **s1**. A terminating wide null character is always appended to the result. The behavior of **wcsncat** is undefined if copying takes place between objects that overlap.

wcsncat returns the value of **s1**.

wcsnchr

Synopsis

```
wchar_t *wcsnchr(const wchar_t *str,
                 size_t n,
                 wchar_t ch);
```

Description

wcsnchr searches not more than **n** wide characters to locate the first occurrence of **c** in the wide string pointed to by **s**. The terminating wide null character is considered to be part of the wide string.

wcsnchr returns a pointer to the located wide character, or a null pointer if **c** does not occur in the string.

wcsncmp

Synopsis

```
int wcsncmp(const wchar_t *s1,
            const wchar_t *s2,
            size_t n);
```

Description

wcsncmp compares not more than **n** wide characters from the array pointed to by **s1** to the array pointed to by **s2**. Characters that follow a null wide character are not compared.

wcsncmp returns an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

wcsncpy

Synopsis

```
wchar_t *wcsncpy(wchar_t *s1,
                 const wchar_t *s2,
                 size_t n);
```

Description

wcsncpy copies not more than **n** wide characters from the array pointed to by **s2** to the array pointed to by **s1**. Wide characters that follow a null wide character in **s2** are not copied. The behavior of **wcsncpy** is undefined if copying takes place between objects that overlap. If the array pointed to by **s2** is a wide string that is shorter than **n** wide characters, null wide characters are appended to the copy in the array pointed to by **s1**, until **n** characters in all have been written.

wcsncpy returns the value of **s1**.

wcsnlen

Synopsis

```
size_t wcsnlen(const wchar_t *s,
               size_t n);
```

Description

this returns the length of the wide string pointed to by **s**, up to a maximum of **n** wide characters. **wcsnlen** only examines the first **n** wide characters of the string **s**.

Note

wcsnlen is an extension commonly found in Linux and BSD C libraries.

wcsnstr

Synopsis

```
wchar_t *wcsnstr(const wchar_t *s1,
                 const wchar_t *s2,
                 size_t n);
```

Description

wcsnstr searches at most **n** wide characters to locate the first occurrence in the wide string pointed to by **s1** of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by **s2**.

wcsnstr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **wcsnstr** returns **s1**.

Note

wcsnstr is an extension commonly found in Linux and BSD C libraries.

wcspbrk

Synopsis

```
wchar_t *wcspbrk(const wchar_t *s1,
                 const wchar_t *s2);
```

Description

wcspbrk locates the first occurrence in the wide string pointed to by **s1** of any wide character from the wide string pointed to by **s2**.

wcspbrk returns a pointer to the wide character, or a null pointer if no wide character from **s2** occurs in **s1**.

wcsrchr

Synopsis

```
wchar_t *wcsrchr(const wchar_t *s,
                 wchar_t c);
```

Description

wcsrchr locates the last occurrence of **c** in the wide string pointed to by **s**. The terminating wide null character is considered to be part of the string.

wcsrchr returns a pointer to the wide character, or a null pointer if **c** does not occur in the wide string.

wcsspn

Synopsis

```
size_t wcsspn(const wchar_t *s1,
              const wchar_t *s2);
```

Description

wcsspn computes the length of the maximum initial segment of the wide string pointed to by **s1** which consists entirely of wide characters from the wide string pointed to by **s2**.

wcsspn returns the length of the segment.

wcsstr

Synopsis

```
wchar_t *wcsstr(const wchar_t *s1,
                const wchar_t *s2);
```

Description

wcsstr locates the first occurrence in the wide string pointed to by **s1** of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by **s2**.

wcsstr returns a pointer to the located wide string, or a null pointer if the wide string is not found. If **s2** points to a wide string with zero length, **wcsstr** returns **s1**.

wcstok

Synopsis

```
wchar_t *wcstok(wchar_t *s1,
                const wchar_t *s2);
```

Description

wcstok A sequence of calls to **wcstok** breaks the wide string pointed to by **s1** into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by **s2**. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator wide string pointed to by **s2** may be different from call to call.

The first call in the sequence searches the wide string pointed to by **s1** for the first wide character that is not contained in the current separator wide string pointed to by **s2**. If no such wide character is found, then there are no tokens in the wide string pointed to by **s1** and **wcstok** returns a null pointer. If such a wide character is found, it is the start of the first token.

wcstok then searches from there for a wide character that is contained in the current wide separator string. If no such wide character is found, the current token extends to the end of the wide string pointed to by **s1**, and subsequent searches for a token will return a null pointer. If such a wude character is found, it is overwritten by a wide null character, which terminates the current token. **wcstok** saves a pointer to the following wide character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Note

wcstok maintains static state and is therefore not reentrant and not thread safe. See **wcstok_r** for a thread-safe and reentrant variant.

wcstok_r

Synopsis

```
wchar_t *wcstok_r(wchar_t *s1,
                  const wchar_t *s2,
                  wchar_t **s3);
```

Description

wcstok_r is a reentrant version of the function **wcstok** where the state is maintained in the object of type **wchar_t *** pointed to by **s3**.

Note

wcstok_r is an extension commonly found in Linux and BSD C libraries.

See Also

wcstok.

wctob

Synopsis

```
int wctob(wint_t c);
```

Description

wctob 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 in the current locale.

Description

this 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**.

wctob_l

Synopsis

```
int wctob_l(wint_t c,
            locale_t loc);
```

Description

wctob_l 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 in locale **loc**.

Description

wctob_l 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**.

wint_t

Synopsis

```
typedef long wint_t;
```

Description

wint_t is an integer type that is unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (WEOF).

wmemccpy

Synopsis

```
wchar_t *wmemccpy(wchar_t *s1,
                  const wchar_t *s2,
                  wchar_t c,
                  size_t n);
```

Description

wmemccpy copies at most **n** wide characters from the object pointed to by **s2** into the object pointed to by **s1**. The copying stops as soon as **n** wide characters are copied or the wide character **c** is copied into the destination object pointed to by **s1**. The behavior of **wmemccpy** is undefined if copying takes place between objects that overlap.

wmemccpy returns a pointer to the wide character immediately following **c** in **s1**, or **NULL** if **c** was not found in the first **n** wide characters of **s2**.

Note

wmemccpy conforms to POSIX.1-2008.

wmemchr

Synopsis

```
wchar_t *wmemchr(const wchar_t *s,
                 wchar_t c,
                 size_t n);
```

Description

wmemchr locates the first occurrence of **c** in the initial **n** characters of the object pointed to by **s**. Unlike **wcschr**, **wmemchr** does *not* terminate a search when a null wide character is found in the object pointed to by **s**.

wmemchr returns a pointer to the located wide character, or a null pointer if **c** does not occur in the object.

wmemcmp

Synopsis

```
int wmemcmp(const wchar_t *s1,
            const wchar_t *s2,
            size_t n);
```

Description

wmemcmp compares the first **n** wide characters of the object pointed to by **s1** to the first **n** wide characters of the object pointed to by **s2**. **wmemcmp** returns an integer greater than, equal to, or less than zero as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

wmemcpy

Synopsis

```
wchar_t *wmemcpy(wchar_t *s1,
                 const wchar_t *s2,
                 size_t n);
```

Description

wmemcpy copies **n** wide characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **wmemcpy** is undefined if copying takes place between objects that overlap.

wmemcpy returns the value of **s1**.

wmemmove

Synopsis

```
wchar_t *wmemmove(wchar_t *s1,
                  const wchar_t *s2,
                  size_t n);
```

Description

wmemmove copies **n** wide characters from the object pointed to by **s2** into the object pointed to by **s1** ensuring that if **s1** and **s2** overlap, the copy works correctly. Copying takes place as if the **n** wide characters from the object pointed to by **s2** are first copied into a temporary array of **n** wide characters that does not overlap the objects pointed to by **s1** and **s2**, and then the **n** wide characters from the temporary array are copied into the object pointed to by **s1**.

wmemmove returns the value of **s1**.

wmempcpy

Synopsis

```
wchar_t *wmempcpy(wchar_t *s1,
                  const wchar_t *s2,
                  size_t n);
```

Description

wmempcpy copies **n** wide characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **wmempcpy** is undefined if copying takes place between objects that overlap.

wmempcpy returns it returns a pointer to the wide character following the last written wide character.

Note

This is an extension found in GNU libc.

wmemset

Synopsis

```
wchar_t *wmemset(wchar_t *s,
                 wchar_t c,
                 size_t n);
```

Description

wmemset copies the value of **c** into each of the first **n** wide characters of the object pointed to by **s**.

wmemset returns the value of **s**.

wstrsep

Synopsis

```
wchar_t *wstrsep(wchar_t **stringp,
                 const wchar_t *delim);
```

Description

wstrsep locates, in the wide string referenced by ***stringp**, the first occurrence of any wide character in the wide string **delim** (or the terminating wide null character) and replaces it with a wide null character. The location of the next character after the delimiter wide character (or NULL, if the end of the string was reached) is stored in ***stringp**. The original value of ***stringp** is returned.

An empty field (that is, a wide character in the string **delim** occurs as the first wide character of ***stringp** can be detected by comparing the location referenced by the returned pointer to a wide null character.

If ***stringp** is initially null, **wstrsep** returns null.

Note

wstrsep is not an ISO C function, but appears in BSD4.4 and Linux.

<wctype.h>

API Summary

Classification functions	
iswalnum	Is character alphanumeric?
iswalpha	Is character alphabetic?
iswblank	Is character blank?
iswcntrl	Is character a control?
iswctype	Determine character type
iswdigit	Is character a decimal digit?
iswgraph	Is character a control?
iswlower	Is character a lowercase letter?
iswprint	Is character printable?
iswpunct	Is character punctuation?
iswspace	Is character a whitespace character?
iswupper	Is character an uppercase letter?
iswxdigit	Is character a hexadecimal digit?
wctype	Construct character class
Conversion functions	
towctrans	Translate character
towlower	Convert uppercase character to lowercase
towupper	Convert lowercase character to uppercase
wctrans	Construct character mapping
Classification functions (extended)	
iswalnum_l	Is character alphanumeric?
iswalpha_l	Is character alphabetic?
iswblank_l	Is character blank?
iswcntrl_l	Is character a control?
iswctype_l	Determine character type
iswdigit_l	Is character a decimal digit?
iswgraph_l	Is character a control?
iswlower_l	Is character a lowercase letter?
iswprint_l	Is character printable?
iswpunct_l	Is character punctuation?

iswspace_l	Is character a whitespace character?
iswupper_l	Is character an uppercase letter?
iswxdigit_l	Is character a hexadecimal digit?
Conversion functions (extended)	
towctrans_l	Translate character
towlower_l	Convert uppercase character to lowercase
towupper_l	Convert lowercase character to uppercase
wctrans_l	Construct character mapping

iswalnum

Synopsis

```
int iswalnum(wint_t c);
```

Description

iswalnum tests for any wide character for which **iswalpha** or **iswdigit** is true.

iswalnum_l

Synopsis

```
int iswalnum_l(wint_t c,
               locale_t loc);
```

Description

iswalnum_l tests for any wide character for which **iswalpha_l** or **iswdigit_l** is true in the locale **loc**.

iswalpha

Synopsis

```
int iswalpha(wint_t c);
```

Description

iswalpha returns true if the wide character **c** is alphabetic. Any character for which **iswupper** or **iswlower** returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

In the C locale, **iswalpha** returns nonzero (true) if and only if **iswupper** or **iswlower** return true for the value of the argument **c**.

iswalpha_l

Synopsis

```
int iswalpha_l(wint_t c,
               locale_t loc);
```

Description

iswalpha_l returns true if the wide character **c** is alphabetic in the locale **loc**. Any character for which **iswupper_l** or **iswlower_l** returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of **iswcntrl_l**, **iswdigit_l**, **iswpunct_l**, or **iswspace_l** is true.

iswblank

Synopsis

```
int iswblank(wint_t c);
```

Description

iswblank tests for any wide character that is a standard blank wide character or is one of a locale-specific set of wide characters for which **iswspace** is true and that is used to separate words within a line of text. The standard blank wide are space and horizontal tab.

In the C locale, **iswblank** returns true only for the standard blank characters.

iswblank_l

Synopsis

```
int iswblank_l(wint_t c,
               locale_t loc);
```

Description

iswblank_l tests for any wide character that is a standard blank wide character in the locale **loc** or is one of a locale-specific set of wide characters for which **iswspace_l** is true and that is used to separate words within a line of text. The standard blank wide are space and horizontal tab.

iswcntrl

Synopsis

```
int iswcntrl(wint_t c);
```

Description

iswcntrl tests for any wide character that is a control character.

iswcntrl_l

Synopsis

```
int iswcntrl_l(wint_t c,
               locale_t loc);
```

Description

iswcntrl_l tests for any wide character that is a control character in the locale **loc**.

iswctype

Synopsis

```
int iswctype(wint_t c,
             wctype_t t);
```

Description

iswctype determines whether the wide character **c** has the property described by **t** in the current locale.

iswctype_l

Synopsis

```
int iswctype_l(wint_t c,
               wctype_t t,
               locale_t loc);
```

Description

iswctype_l determines whether the wide character **c** has the property described by **t** in the locale **loc**.

iswdigit

Synopsis

```
int iswdigit(wint_t c);
```

Description

iswdigit tests for any wide character that corresponds to a decimal-digit character.

iswdigit_l

Synopsis

```
int iswdigit_l(wint_t c,
               locale_t loc);
```

Description

iswdigit_l tests for any wide character that corresponds to a decimal-digit character in the locale **loc**.

iswgraph

Synopsis

```
int iswgraph(wint_t c);
```

Description

iswgraph tests for any wide character for which **iswprint** is true and **iswspace** is false.

iswgraph_l

Synopsis

```
int iswgraph_l(wint_t c,
               locale_t loc);
```

Description

iswgraph_l tests for any wide character for which **iswprint** is true and **iswspace** is false in the locale **loc**.

iswlower

Synopsis

```
int iswlower(wint_t c);
```

Description

iswlower tests for any wide character that corresponds to a lowercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

iswlower_l

Synopsis

```
int iswlower_l(wint_t c,
               locale_t loc);
```

Description

iswlower_l tests for any wide character that corresponds to a lowercase letter in the locale **loc** or is one of a locale-specific set of wide characters for which none of **iswcntrl_l**, **iswdigit_l**, **iswpunct_l**, or **iswspace_l** is true.

iswprint

Synopsis

```
int iswprint(wint_t c);
```

Description

iswprint returns nonzero (true) if and only if the value of the argument **c** is any printing character.

iswprint_l

Synopsis

```
int iswprint_l(wint_t c,
               locale_t loc);
```

Description

iswprint_l returns nonzero (true) if and only if the value of the argument **c** is any printing character in the locale **loc**.

iswpunct

Synopsis

```
int iswpunct(wint_t c);
```

Description

iswpunct tests for any printing wide character that is one of a locale-specific set of punctuation wide characters for which neither **iswspace** nor **iswalnum** is true.

iswpunct_l

Synopsis

```
int iswpunct_l(wint_t c,
               locale_t loc);
```

Description

iswpunct_l tests for any printing wide character that is one of a locale-specific set of punctuation wide characters in locale **loc** for which neither **iswspace_l** nor **iswalnum_l** is true.

iswspace

Synopsis

```
int iswspace(wint_t c);
```

Description

iswspace tests for any wide character that corresponds to a locale-specific set of white-space wide characters for which none of **iswalnum**, **iswgraph**, or **iswpunct** is true.

iswspace_l

Synopsis

```
int iswspace_l(wint_t c,
               locale_t loc);
```

Description

iswspace_l tests for any wide character that corresponds to a locale-specific set of white-space wide characters in the locale **loc** for which none of **iswalnum**, **iswgraph_l**, or **iswpunct_l** is true.

iswupper

Synopsis

```
int iswupper(wint_t c);
```

Description

iswupper tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

iswupper_l

Synopsis

```
int iswupper_l(wint_t c,
               locale_t loc);
```

Description

iswupper_l tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters in the locale **loc** for which none of **iswcntrl_l**, **iswdigit_l**, **iswpunct_l**, or **iswspace_l** is true.

iswxdigit

Synopsis

```
int iswxdigit(wint_t c);
```

Description

iswxdigit tests for any wide character that corresponds to a hexadecimal digit.

iswxdigit_l

Synopsis

```
int iswxdigit_l(wint_t c,
                locale_t loc);
```

Description

iswxdigit_l tests for any wide character that corresponds to a hexadecimal digit in the locale **loc**.

towctrans

Synopsis

```
wint_t towctrans(wint_t c,
                 wctrans_t t);
```

Description

towctrans maps the wide character **c** using the mapping described by **t** in the current locale.

towctrans_l

Synopsis

```
wint_t towctrans_l(wint_t c,
                   wctrans_t t,
                   locale_t loc);
```

Description

towctrans_l maps the wide character **c** using the mapping described by **t** in the current locale.

towlower

Synopsis

```
wint_t towlower(wint_t c);
```

Description

towlower converts an uppercase letter to a corresponding lowercase letter.

If the argument **c** is a wide character for which **iswupper** is true and there are one or more corresponding wide characters, in the current locale, for which **iswlower** is true, **towlower** returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, **c** is returned unchanged.

towlower_l

Synopsis

```
wint_t towlower_l(wint_t c,
                  locale_t loc);
```

Description

towlower_l converts an uppercase letter to a corresponding lowercase letter in locale **loc**.

If the argument **c** is a wide character for which **iswupper_l** is true and there are one or more corresponding wide characters, in the locale **loc**, for which **iswlower_l** is true, **towlower_l** returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, **c** is returned unchanged.

towupper

Synopsis

```
wint_t towupper(wint_t c);
```

Description

towupper converts a lowercase letter to a corresponding uppercase letter.

If the argument **c** is a wide character for which **iswlower** is true and there are one or more corresponding wide characters, in the current current locale, for which **iswupper** is true, **towupper** returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, **c** is returned unchanged.

towupper_l

Synopsis

```
wint_t towupper_l(wint_t c,
                  locale_t loc);
```

Description

towupper_l converts a lowercase letter to a corresponding uppercase letter in locale **loc**.

If the argument **c** is a wide character for which **iswlower_l** is true and there are one or more corresponding wide characters, in the locale **loc**, for which **iswupper_l** is true, **towupper_l** returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, **c** is returned unchanged.

wctrans

Synopsis

```
wctrans_t wctrans(const char *property);
```

Description

wctrans constructs a value of type **wctrans_t** that describes a mapping between wide characters identified by the string argument **property**.

If **property** identifies a valid mapping of wide characters in the current locale, **wctrans** returns a nonzero value that is valid as the second argument to **towctrans**; otherwise, it returns zero.

Note

The only mappings supported are `"tolower"` and `"toupper"`.

wctrans_l

Synopsis

```
wctrans_t wctrans_l(const char *property,
                    locale_t loc);
```

Description

wctrans_l constructs a value of type **wctrans_t** that describes a mapping between wide characters identified by the string argument **property** in locale **loc**.

If **property** identifies a valid mapping of wide characters in the locale **loc**, **wctrans_l** returns a nonzero value that is valid as the second argument to **towctrans_l**; otherwise, it returns zero.

Note

The only mappings supported are `"tolower"` and `"toupper"`.

wctype

Synopsis

```
wctype_t wctype(const char *property);
```

Description

wctype constructs a value of type **wctype_t** that describes a class of wide characters identified by the string argument **property**.

If **property** identifies a valid class of wide characters in the current locale, **wctype** returns a nonzero value that is valid as the second argument to **iswctype**; otherwise, it returns zero.

Note

The only mappings supported are `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"`, and `"xdigit"`.

<xlocale.h>

API Summary

Functions	
duplocale	Duplicate current locale data
freelocale	Free a locale
localeconv_l	Get locale data
newlocale	Create a new locale

duplocale

Synopsis

```
locale_t duplocale(locale_t loc);
```

Description

duplocale duplicates the locale object referenced by **loc**.

If there is insufficient memory to duplicate **loc**, **duplocale** returns **NULL** and sets **errno** to **ENOMEM** as required by POSIX.1-2008.

Duplicated locales must be freed with **freelocale**.

This is different behavior from the GNU glibc implementation which makes no mention of setting **errno** on failure.

Note

This extension is derived from BSD, POSIX.1, and glibc.

freelocale

Synopsis

```
int freelocale(locale_t loc);
```

Description

freelocale frees the storage associated with **loc**.

freelocale zero on success, 1 on error.

localeconv_l

Synopsis

```
localeconv_l(locale_t loc);
```

Description

localeconv_l returns a pointer to a structure of type **lconv** with the corresponding values for the locale **loc** filled in.

newlocale

Synopsis

```
locale_t newlocale(int category_mask,
                   const char *locale,
                   locale_t base);
```

Description

newlocale creates a new locale object or modifies an existing one. If the base argument is **NULL**, a new locale object is created.

category_mask specifies the locale categories to be set or modified. Values for **category_mask** are constructed by a bitwise-inclusive OR of the symbolic constants **LC_CTYPE_MASK**, **LC_NUMERIC_MASK**, **LC_TIME_MASK**, **LC_COLLATE_MASK**, **LC_MONETARY_MASK**, and **LC_MESSAGES_MASK**.

For each category with the corresponding bit set in **category_mask**, the data from the locale named by **locale** is used. In the case of modifying an existing locale object, the data from the locale named by **locale** replaces the existing data within the locale object. If a completely new locale object is created, the data for all sections not requested by **category_mask** are taken from the default locale.

The locales C and POSIX are equivalent and defined for all settings of **category_mask**:

If **locale** is **NULL**, then the C locale is used. If **locale** is an empty string, **newlocale** will use the default locale.

If **base** is **NULL**, the current locale is used. If **base** is **LC_GLOBAL_LOCALE**, the global locale is used.

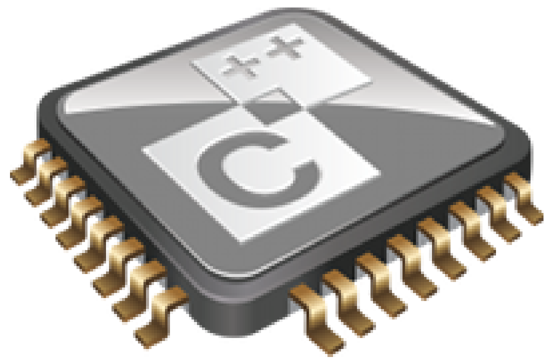
If **mask** is **LC_ALL_MASK**, **base** is ignored.

Note

POSIX.1-2008 does not specify whether the locale object pointed to by **base** is modified or whether it is freed and a new locale object created.

Implementation

The category mask **LC_MESSAGES_MASK** is not implemented as POSIX messages are not implemented.



C++ Library User Guide

SEGGER Embedded Studio provides a limited C++ library suitable for use in an embedded application.

Standard library

The following C++ standard header files are provided in `$(StudioDir)/include`:

File	Description
<cassert>	C++ wrapper on **assert.h**.
<cctype>	C++ wrapper on **ctype.h**.
<cerrno>	C++ wrapper on **errno.h**.
<cfloat>	C++ wrapper on **float.h**.
<ciso646>	C++ wrapper on **iso646.h**.
<climits>	C++ wrapper on **limits.h**.
<clocale>	C++ wrapper on **locale.h**.
<cmath>	C++ wrapper on **math.h**.
<csetjmp>	C++ wrapper on **setjmp.h**.
<cstdarg>	C++ wrapper on **stdarg.h**.
<cstddef>	C++ wrapper on **stddef.h**.
<cstdio>	C++ wrapper on **stdio.h**.
<cstdlib>	C++ wrapper on **stdlib.h**.

<cstring>	C++ wrapper on **string.h**.
<ctime>	C++ wrapper on **time.h**.
<cwchar>	C++ wrapper on **wchar.h**.
<cwctype>	C++ wrapper on **wctype.h**.
<exception>	Definitions for exceptions.
<new>	Types and definitions for placement new and delete.
<typeinfo>	Definitions for RTTI. *Note that this file is licensed under the GPL*.

It's worth mentioning again: to use exceptions or RTTI requires header files and or library code to be linked into your application that is licensed under the GPL.

Standard template library

The C++ STL functionality of STLPort 5.1.0 is provided in SEGGER Embedded Studio. To use STLPort you must put `$(StudioDir)/include/stlport` as the first entry in the **User Include Directories** project property. The STLPort is configured to *not* support long doubles and iostreams. The following STLPort header files are supported (not including the above list of standard C++ header files)

<algorithm>	<bitset>	<deque>
<functional>	<hash_map>	<hash_set>
<iterator>	<limits>	<list>
<locale>	<map>	<memory>
<numeric>	<queue>	<set>
<stack>	<stdexcept>	<string>
<utility>	<valarray>	<vector>

Subset API reference

This section contains a subset reference to the SEGGER Embedded Studio C++ library.

<new> - memory allocation

The header file <**new**> defines functions for memory allocation.

Functions	
set_new_handler	Establish a function which is called when memory allocation fails.
Operators	
operator delete	Heap storage deallocators operator.
operator new	Heap storage allocators operator.

operator delete

Synopsis

```
void operator delete(void *ptr) throw();
```

```
void operator delete[](void *ptr) throw();
```

Description

operator delete deallocates space of an object.

operator delete will do nothing if **ptr** is null. If **ptr** is not null then it should have been returned from a call to **operator new**.

operator delete[] has the same behaviour as **operator delete** but is used for array deallocation.

Portability

Standard C++.

operator new

Synopsis

```
void *operator new(size_t size) throw();
```

```
void *operator new[](size_t size) throw();
```

Description

operator new allocates space for an object whose size is specified by **size** and whose value is indeterminate.

operator new returns a null pointer if the space for the object cannot be allocated from free memory; if space for the object can be allocated, **operator new** returns a pointer to the start of the allocated space.

operator new[] has the same behaviour as **operator new** but is used for array allocation.

Portability

The implementation is not standard. The standard C++ implementation should throw an exception if memory allocation fails.

set_new_handler

Synopsis

```
typedef void (*new_handler)();
```

```
new_handler set_new_handler(new_handler) throw();
```

Description

set_new_handler establishes a **new_handler** function.

set_new_handler establishes a **new_handler** function that is called when **operator new** fails to allocate the requested memory. If the **new_handler** function returns then **operator new** will attempt to allocate the memory again. The **new_handler** function can throw an exception to implement standard C++ behaviour for memory allocation failure.

Portability

Standard C++.

Utilities Reference

Compiler driver

This section describes the switches accepted by the compiler driver, **cc**. The compiler driver is capable of controlling compilation by all supported language compilers and the final link by the linker. It can also construct libraries automatically.

In contrast to many compilation and assembly language development systems, with you don't invoke the assembler or compiler directly. Instead you'll normally use the compiler driver **cc** as it provides an easy way to get files compiled, assembled, and linked. This section will introduce you to using the compiler driver to convert your source files to object files, executables, or other formats.

We recommend that you use the compiler driver rather than use the assembler or compiler directly because there the driver can assemble multiple files using one command line and can invoke the linker for you too. There is no reason why you should not invoke the assembler or compiler directly yourself, but you'll find that typing in all the required options is quite tedious-and why do that when **cc** will provide them for you automatically?

File naming conventions

The compiler driver uses file extensions to distinguish the language the source file is written in. The compiler driver recognizes the extension **.c** as C source files, **.cpp**, **.cc** or **.cxx** as C++ source files, **.s** and **.asm** as assembly code files.

The compiler driver recognizes the extension **.o** as object files, **.a** as library files, **.ld** as linker script files and **.xml** as special-purpose XML files.

We strongly recommend that you adopt these extensions for your source files and object files because you'll find that using the tools is much easier if you do.

C language files

When the compiler driver finds a file with a **.c** extension, it runs the C compiler to convert it to object code.

C++ language files

When the compiler driver finds a file with a **.cpp** extension, it runs the C++ compiler to convert it to object code.

Assembly language files

When the compiler driver finds a file with a **.s** or **.asm** extension, it runs the C preprocessor and then the assembler to convert it to object code.

Object code files

When the compiler driver finds a file with a **.o** or **.a** extension, it passes it to the linker to include it in the final application.

Command-line options

This section describes the command-line options accepted by the SEGGER Embedded Studio compiler driver.

-ansi (Warn about potential ANSI problems)

Syntax

-ansi

Description

Warn about potential problems that conflict with the relevant ANSI or ISO standard for the files that are compiled.

-ar (Archive output)

Syntax

-ar

Description

This switch instructs the compiler driver to archive all output files into a library. Using **-ar** implies **-c.**

Example

The following command compiles **file1.c**, **file2.asm**, and **file3.c** to object code and archives them into the library file **libfunc.a** together with the object file **file4.o**.

```
cc -ar file1.c file2.asm file3.c file4.o -o libfunc.a
```

-arch (Set ARM architecture)

Syntax

-arch=*a*

Description

Specifies the version of the instruction set to generate code for. The options are:

-arch=v4T ARM7TDMI and ARM920T
-arch=v5TE ARM9E, Feroceon and XScale
-arch=v6 ARM11
-arch=v6M Cortex-M0 and Cortex-M1
-arch=v7A Cortex-A8 and Cortex-A9
-arch=v7M Cortex-M3
-arch=v7EM Cortex-M4
-arch=v7R Cortex-R4

Example

To force compilation for V7A architecture you would use:

```
cc -arch=v7A
```

-be (Big Endian)

Syntax

-be

Description

Generate code for a big endian target.

-c (Compile to object code, do not link)

Syntax

-c

Description

All named files are compiled to object code modules, but are not linked. You can use the **-o** option to name the output if you just supply one input filename.

Example

The following command compiles **file1.c** and **file4.c** to produce the object files **file1.o** and **file4.o**.

```
cc -c file1.c file4.c
```

The following command compiles **file1.c** and produces the object file **obj/file1.o**.

```
cc -c file.c -o obj/file1.o
```

-d (Define linker symbol)

Syntax

-d*name*=*value*

Description

You can define linker symbols using the **-d** option. The symbol definitions are passed to linker.

Example

The following defines the symbol, **STACK_SIZE** with a value of 512.

```
-dSTACK_SIZE=512
```

-D (Define macro symbol)

Syntax

-D*name*
-D*name*=*value*

Description

You can define preprocessor macros using the **-D** option. The macro definitions are passed on to the respective language compiler which is responsible for interpreting the definitions and providing them to the programmer within the language.

The first form above defines the macro *name* but without an associated replacement value, and the second defines the same macro with the replacement value *value*.

Example

The following defines two macros, **SUPPORT_FLOAT** with a value of 1 and **LITTLE_ENDIAN** with no replacement value.

```
-DSUPPORT_FLOAT=1 -DLITTLE_ENDIAN
```

-e (Set entry point symbol)

Syntax

-e*name*

Description

Linker option to set the entry point symbol to be *name*. The debugger will start execution from this symbol.

-E (Preprocess)

Syntax

-E

Description

This option preprocesses the supplied file and outputs the result to the standard output.

Example

The following preprocesses the file **file.c** supplying the macros, **SUPPORT_FLOAT** with a value of 1 and **LITTLE_ENDIAN**.

```
-E -DSUPPORT_FLOAT=1 -DLITTLE_ENDIAN file.c
```

-exceptions (Enable C++ Exception Support)

Syntax

-exceptions

Description

Enables C++ exceptions to be compiled.

-fabi (Floating Point Code Generation)

Syntax

-fabi=*a*

Description

Specifies the type of floating point code generation. The options are:

-fabi=SoftFP FPU instructions are generated, CPU registers are used for floating point parameters.
-fabi=Hard FPU instructions are generated, FPU registers are used for floating point parameters.

-fpu (Set ARM FPU)

Syntax

-fpu=*a*

Description

Specifies the floating point unit to generate code for when the **fpabi** option has been supplied. The options are:

-fpu=VFP generate FPU instructions for ARM9 and ARM11
-fpu=VFPv3-D32 generate FPU instructions for CortexA
-fpu=VFPv3-D16 generate FPU instructions for CortexR
-fpu=FPv4-SP-D16 generate FPU instructions for CortexM4

-F (Set output format)

Syntax

-F*fmt*

Description

The **-F** option instructs the compiler driver to generate an additional output file in the format *fmt*. The compiler driver supports the following formats:

-Fbin Create a .bin file
-Fhex Create a .hex file
-Fsrec Create a .srec file

The compiler driver will always output a **.elf** file as specified with the **-o** option. The name of the additional output file is the same as the **.elf** file with the file extension changed.

For example

```
cc file.c -o file.elf -Fbin
```

will generate the files **file.elf** and **file.bin**.

-g (Generate debugging information)

Syntax

-g

Description

The **-g** option instructs the compiler and assembler to generate source level debugging information for the debugger to use.

-g1 (Generate minimal debugging information)

Syntax

-g1

Description

The **-g1** option instructs the compiler to generate debugging information that enables the debugger to be able to backtrace only.

-help (Display help information)

Syntax

-help

Description

Displays a short summary of the options accepted by the compiler driver.

-io (Select I/O library implementation)

Syntax

-io=*i*

Description

This option specifies the I/O library implementation that is included in the linked image. The options are:

-io=d I/O library is implemented using debugIO e.g calls to **printf** will call **debug_printf**.
-io=t I/O library is implemented on the target, debugIO is not used.
-io=t+d I/O library is implemented on the target, debugIO is not used but debugIO is enabled.

-I (Define user include directories)

Syntax

-I*directory*

Description

In order to find include files the compiler driver arranges for the compilers to search a number of standard directories. You can add directories to the search path using the **-I** switch which is passed on to each of the language processors.

You can specify more than one include directory by separating each directory component with either a comma or semicolon.

-I- (Exclude standard include directories)

Syntax

-I-

Description

Usually the compiler and assembler search for include files in the standard include directory created when the product is installed. If for some reason you wish to exclude these system locations from being searched when compiling a file, the **-I-** option will do this for you.

-J (Define system include directories)

Syntax

-J*directory*

Description

The **-J** option adds *directory* to the end of the list of directories to search for source files included (using triangular brackets) by the `#include` preprocessor command.

You can specify more than one include directory by separating each directory component with either a comma or semicolon in the property

-K (Keep linker symbol)

Syntax

-K*name*

Description

The linker removes unused code and data from the output file. This process is called *deadstripping*. To prevent the linker from deadstripping unreferenced code and data you wish to keep, you must use the **-K** command line option to force inclusion of symbols.

Example

If you have a C function, **contextSwitch** that must be kept in the output file (and which the linker will normally remove), you can force its inclusion using:

```
-KcontextSwitch
```

-L (Set library directory path)

Syntax

-L*dir*

Description

Sets the library directory to *dir*. If **-L** is not specified on the command line, the default location to search for libraries is set to **$(InstallDir)/lib**.

-l- (Do not link standard libraries)

Syntax

-l-

Description

The **-l** option instructs the compiler driver not to link standard libraries. If you use this option you must supply your own library functions or libraries.

-make (Make-style build)

Syntax

-make

Description

The **-make** option avoids build steps based on the modification date of the output file and modification date of the input file and its dependencies.

-M (Display linkage map)

Syntax

-M

Description

The **-M** option prints a linkage map named the same as the linker output file with the **.map** file extension.

-n (Dry run, no execution)

Syntax

-n

Description

When **-n** is specified, the compiler driver processes options as usual, but does not execute any subprocesses to compile, assemble, archive or link applications.

-nostderr (No stderr output)

Syntax

-nostderr

Description

When **-nostderr** is specified, any stderr output of subprocesses is redirected to stdout.

-o (Set output file name)

Syntax

-o *filename*

Description

The **-o** option instructs the compiler driver to write linker or archiver output to *filename*.

-oabi (Use oabi compiler)

Syntax

-oabi

Description

The **-oabi** option instructs the compiler driver to generate code and link libraries for the legacy GCC ARM ABI.

-O (Optimize output)

Syntax

-O*x*

Description

Pass the optimization option **-O***x* to the compiler and select library variant. The following options are supported:

-O0 No optimization, use libraries built with **-O1.**
-O1 Level 1 optimization, use libraries built with **-O1.**
-O2 Level 2 optimization, use libraries built with **-O1.**
-O3 Level 3 optimization, use libraries built with **-O1.**
-Os Optimize for size, use libraries built with **-Os.**

-printf (Select printf capability)

Syntax

-printf=*c*

Description

The **-printf** option selects the printf capability for the linked executable. The options are:

-printf=i integer is supported
-printf=li long integer is supported
-printf=ll long long integer is supported
-printf=f floating point is supported
-printf=wp width and precision is supported

-rtti (Enable C++ RTTI Support)

Syntax

-rtti

Description

Enables C++ run-time type information to be compiled.

-R (Set section name)

Syntax

-R *x name*

Description

These options name the default name of the sections generated by the compiler/assembler to be *name*. The options are:

-Rc *name* change the default name of the code section
-Rd *name* change the default name of the data section
-Rk *name* change the default name of the const section
-Rz *name* change the default name of the bss section

-scanf (Select scanf capability)

Syntax

-scanf= *c*

Description

The **-scanf** option selects the scanf capability for the linked executable. The options are:

-scanf=i integer is supported
-scanf=li long integer is supported
-scanf=ll long long integer is supported
-scanf=f floating point is supported
-scanf=wp %[...] and %[^...] character class is supported

-sd (Treat double as float)

Syntax

-sd

Description

The **-sd** option instructs the compiler to compile double as float and selects the appropriate library for linking.

-Thumb (Generate Thumb code)

Syntax

-Thumb

Description

The **-Thumb** option instructs the compiler to generate Thumb code rather than ARM code and link in Thumb libraries. This option is NOT needed for Cortex-M architectures.

-v (Verbose execution)

Syntax

-v

Description

The **-v** switch displays command lines executed by the compiler driver.

-w (Suppress warnings)

Syntax

-w

Description

This option instructs the compiler, assembler, and linker not to issue any warnings.

-we (Treat warnings as errors)

Syntax

-we

Description

This option directs the compiler, assembler, and linker to treat all warnings as errors.

-Wa (Pass option to tool)

Syntax

-W*tool option*

Description

The **-W** command-line option passes *option* directly to the specified *tool*. Supported tools are

- **-Wa** pass option to assembler
- **-Wc** pass option to compiler
- **-Wl** pass option to linker

Example

The following example passes the (compiler specific) -version option to the compiler

```
cc  -Wc-version
```

-x (Specify file types)

Syntax

-x *type*

Description

The **-x** option causes the compiler driver to treat subsequent files to be of the following file type

-xa archives/libraries
-xasm assembly code files
-xc C code files
-xc++ C++ code files
-xld linker script files
-xo object code files

Example

The following command line enables an assembly code file with the extension **.arm** to be assembled.

```
cc -xasm a.arm
```

-y (Use project template)

Syntax

-y *t*

Description

If required this option must be the first option on the command line. It instantiates a project template type from the installed packages. The files and common project properties of the project template are used by the compiler driver. Project configurations are not supported by the compiler driver, use emBuild if you require project configurations.

Example

The following command builds an executable based on the STM32_EXE project template.

```
cc -ySTM32_EXE -zTarget=STM32F100C4 file.c -o file.elf
```

-z (Set project property)

Syntax

-z *p* = *v*

Description

Sets the value of the project property *p* to the value *v*.

Example

The following command compiles the file arguments and puts the resulting object files into the directory **objects**.

```
cc -c file1.c file2.c -zbuild_output_directory=objects
```

Command-Line Project Builder

emBuild is a program used to build your software from the command line without using **SEGGER** Embedded Studio. You can, for example, use **emBuild** for nightly (automated) builds, production builds, and batch builds.

Building with a SEGGER Embedded Studio project file

You can specify a SEGGER Embedded Studio project file:

Syntax

emBuild [*options*] *project-file*

You must specify a configuration to build using **-config.** For instance:

```
emBuild -config "V5T Thumb LE Release" arm.emProject
```

The above example uses the configuration **V5T Thumb LE Release** to build all projects in the solution contained in **arm.emProject**.

To build a specific project that is in a solution, you can specify it using the **-project** option. For example:

```
emBuild -config "V5T Thumb LE Release" -project "libm" libc.emProject
```

This example will use the configuration **V5T Thumb LE Release** to build the project **libm** that is contained in **libc.emProject**.

If your project file imports other project files (using the <import> mechanism), when denoting projects you must specify the solution names as a comma-separated list in parentheses after the project name:

```
emBuild -config "V5T Thumb LE Release" -project "libc(C Library)" arm.emProject
```

libc(C Library) specifies the **libc** project in the **C Library** solution that has been imported by the project file **arm.emProject**.

To build a specific solution that has been imported from other project files, you can use the **-solution** option. This option takes the solution names as a comma-separated list. For example:

```
emBuild -config "ARM Debug" -solution "ARM Targets,EB55" arm.emProject
```

In this example, **ARM Targets,EB55** specifies the **EB55** solution imported by the **ARM Targets** solution, which was itself imported by the project file **arm.emProject**.

You can do a batch build using the **-batch** option:

```
emBuild -config "ARM Debug" -batch libc.emProject
```

This will build the projects in **libc.emProject** that are marked for batch build in the configuration **ARM Debug**.

By default, a *make-style* build will be donei.e., the dates of input files are checked against the dates of output files, and the build is avoided if the output is up to date. You can force a complete build by using the **-rebuild** option. Alternatively, to remove all output files, use the **-clean** option.

To see the commands being used in the build, use the **-echo** option. To also see why commands are being executed, use the **-verbose** option. You can see what commands will be executed, without executing them, by using the **-show** option.

Building without a SEGGER Embedded Studio project file

To use **emBuild** without a SEGGER Embedded Studio project, specify the name of an installed project template, the name of the project, and the files to build. For example:

```
emBuild -config  -template LM3S_EXE -project myproject -file main.c
```

Or, instead of a template, you can specify a project type:

```
emBuild -config  -type "Library" -project myproject -file main.c
```

You can specify project properties with the **-property** option:

```
emBuild  -property Target=LM3S811
```

Command-line options

This section describes the command-line options accepted by emBuild.

-batch (Batch build)

Syntax

-batch

Description

Perform a batch build.

-config (Select build configuration)

Syntax

-config *name*

Description

Specify the configuration for a build. If the configuration *name* can't be found, emBuild will list the available configurations.

-clean (Remove output files)

Syntax

-clean

Description

Remove all output files resulting from the build process.

-D (Define macro)

Syntax

-D *macro=value*

Description

Define a SEGGER Embedded Studio macro value for the build process.

-echo (Show command lines)

Syntax

-echo

Description

Show the command lines as they are executed.

-file (Build a named file)

Syntax

-file *name*

Description

Build the file *name*. Use with **-template** or **-type.**

-packagesdir (Specify packages directory)

Syntax

-packagesdir *dir*

Description

Override the default value of the **$(PackagesDir)** macro.

-project (Specify project to build)

Syntax

-project *name*

Description

Specify the name of the project to build. When used with a project file, if emBuild can't find the specified project, the names of available projects are listed.

-property (Set project property)

Syntax

-project *name=value*

Description

Specify the value of a project property use with **-template** or **-type.** If emBuild cannot find the specified property, a list of the properties is shown.

-rebuild (Always rebuild)

Syntax

-rebuild

Description

Always execute the build commands.

-show (Dry run, don't execute)

Syntax

-show

Description

Show the command lines that would be executed, but do not execute them.

-solution (Specify solution to build)

Syntax

-solution *name*

Description

Specify the name of the solution to build. If emBuild cannot find the given solution, the valid solution names are listed.

-studiodir (Specify SEGGER Embedded Studio directory)

Syntax

-studiodir *name*

Description

Override the default value of the **$(StudioDir)** macro.

-template (Specify project template)

Syntax

-template *name*

Description

Specify the project template to use. If emBuild cannot find the specified template then a list of template names is shown.

-time (Time the build)

Syntax

-time

Description

Show the time taken for the build.

-threadnum (Specify number of build threads)

Syntax

-threadnum *n*

Description

Specify the number of build threads to use for the build. The default is zero which will use the number of processor cores on your machine.

-type (Specify project type)

Syntax

-type *name*

Description

Specify the project type to use. If emBuild cannot find the specified project type then a list of project type names is shown.

-verbose (Show build information)

Syntax

-verbose

Description

Show extra information relating to the build process.

Command-Line Scripting

emScript is a program that allows you to run SEGGER Embedded Studio's JavaScript (ECMAScript) interpreter from the command line.

The primary purpose of **emScript** is to facilitate the creation of platform-independent build scripts.

Syntax

emScript [*options*] *file*

Command-line options

This section describes the command-line options accepted by emScript.

-define (Define global variable)

Syntax

-define *variable*=*value*

Description

-help (Show usage)

Syntax

-help

Description

Display usage information and command line options.

-load (Load script file)

Syntax

-load *path*

Description

Loads the script file *path*.

-define (Verbose output)

Syntax

-verbose

Description

Produces verbose output.

emScript classes

emScript provides the following predefined classes:

- **BinaryFile**
- **CWSys**
- **ElfFile**
- **WScript**

Example uses

The following example demonstrates using **emScript** to increment a build number:

First, add a JavaScript file to your project called `incbuild.js` containing the following code:

```
function incbuild()
{
  var file = "buildnum.h"
  var text = "#define BUILDNUMBER "
  var s = CWSys.readStringFromFile(file);
  var n;
  if (s == undefined)
    n = 1;
  else
    n = eval(s.substring(text.length)) + 1;
  CWSys.writeStringToFile(file, text + n);
}

// Executed when script loaded.
incbuild();
```

Add a file called `getbuildnum.h` to your project containing the following code:

```
#ifndef GETBUILDNUM_H
#define GETBUILDNUM_H

unsigned getBuildNumber();

#endif
```

Add a file called `getbuildnum.c` to your project containing the following code:

```
#include "getbuildnum.h"
#include "buildnum.h"

unsigned getBuildNumber()
{
  return BUILDNUMBER;
}
```

Now, to combine these:

Set the **Build Options** > **Always Rebuild** project property of `getbuildnum.c` to **Yes**.

Set the **User Build Step Options** > **Pre-Compile Command** project property of `getbuildnum.c` to **"$(StudioDir)/bin/emScript" -load "$(ProjectDir)/incbuild.js"**.

Embed

Embed is a program that converts a binary file into a C/C++ array definition.

The primary purpose of the **Embed** tool is to provide a simple method of embedding files into an application. This may be useful if you want to include firmware images, bitmaps, etc. in your application without having to read them first from an external source.

Syntax

embed *variable_name input_file output_file*

variable_name is the name of the C/C++ array to be initialised with the binary data.

input_file is the path to the binary input file.

output_file is the path to the C/C++ source file to generate.

Example

To convert a binary file *image.bin* to a C/C++ file called *image.h*:

```
embed img image.bin image.h
```

This will generate the following output in *image.h*:

```
static const unsigned char img[] = {
  0x5B, 0x95, 0xA4, 0x56, 0x16, 0x5F, 0x2D, 0x47,
  0xC5, 0x04, 0xD4, 0x8D, 0x73, 0x40, 0x31, 0x66,
  0x3E, 0x81, 0x90, 0x39, 0xA3, 0x8E, 0x22, 0x37,
  0x3C, 0x63, 0xC8, 0x30, 0x90, 0x0C, 0x54, 0xA4,
  0xA2, 0x74, 0xC2, 0x8C, 0x1D, 0x56, 0x57, 0x05,
  0x45, 0xCE, 0x3B, 0x92, 0xAD, 0x0B, 0x2C, 0x39,
  0x92, 0x59, 0xB9, 0x9D, 0x01, 0x30, 0x59, 0x9F,
  0xC5, 0xEA, 0xCE, 0x35, 0xF6, 0x4B, 0x05, 0xBF
};
```

Header file generator

The command line program **mkhdr** generates a C or C++ header file from a SEGGER Embedded Studio memory map file.

Using the header generator

For each register definition in the memory map file a corresponding **#define** is generated in the header file. The **#define** is named the same as the register name and is defined as a volatile pointer to the address.

The type of the pointer is derived from the size of the register. A four-byte register generates an unsigned long pointer. A two-byte register generates an unsigned short pointer. A one-byte register will generates an unsigned char pointer.

If a register definition in the memory map file has bitfields then preprocessor symbols are generated for each bitfield. Each bitfield will have two preprocessor symbols generated, one representing the mask and one defining the start bit position. The bitfield preprocessor symbol names are formed by prepending the register name to the bitfield name. The mask definition has **_MASK** appended to it and the start definition has **_BIT** appended to it.

For example consider the following definitions in the the file **memorymap.xml**.

```
<RegisterGroup start="0xFFFFF000" name="AIC">
  <Register start="+0x00" size="4" name="AIC_SMR0">
    <BitField size="3" name="PRIOR" start="0" />
    <BitField size="2" name="SRCTYPE" start="5" />
  </Register>
  ...
```

We can generate the header file associated with this file using:

```
mkhdr memorymap.xml memorymap.h
```

This generates the following definitions in the file **memorymap.h**.

```
#define AIC_SMR0 (*(volatile unsigned long *)0xFFFFF000)
#define AIC_SMR0_PRIOR_MASK 0x7
#define AIC_SMR0_PRIOR_BIT 0
#define AIC_SMR0_SRCTYPE_MASK 0x60
#define AIC_SMR0_SRCTYPE_BIT 5
```

These definitions can be used in the following way in a C/C++ program:

Reading a register

```
unsigned r = AIC_SMR0;
```

Writing a register

```
AIC_SMR0 = (priority << AIC_SMR0_PRIOR_BIT) | (srctype << AIC_SMR0_SRCTYPE_BIT);
```

Reading a bitfield

```
unsigned srctype = (AIC_SMR0 & AIC_SMR0_SRCTYPE_MASK) >> AIC_SMR0_SRCTYPE_BIT;
```

Writing a bitfield

```
AIC_SMR0 = (AIC_SMR0 & ~AIC_SMR0_SRCTYPE_MASK) | ((srctype & AIC_SMR0_SRCTYPE_MASK) << AIC_SMR0_SRCTYPE_BIT)
```

Command line options

This section describes the command line options accepted by the header file generator.

Syntax

mkhdr *inputfile outputfile targetname* [*option*]

inputfile is the name of the source SEGGER Embedded Studio memory map file. **outputfile** is the the name of the file to write.

-regbaseoffsets (Use offsets from peripheral base)

Syntax

-regbaseoffsets

Description

Instructs the header generator to include offsets of registers from the peripheral base.

-nobitfields (Inhibit bitfield macros)

Syntax

-nobitfields

Description

Instructs the header generator not to generate any definitions for bitfields.

Linker script file generator

The command line program **mkld** generates a GNU ld linker script from a SEGGER Embedded Studio memory map or section placement file.

Syntax

mkld *-memory-map-file inputfile outputfile* [*options*]

inputfile is the name of the SEGGER Embedded Studio memory map file to generate the ld script from.
outputfile is the the name of the ld script file to write.

Command-line options

This section describes the command-line options accepted by *mkld*.

-check-segment-overflow

Syntax

-check-segment-overflow

Description

Add checks for memory segment overflow to the linker script.

-memory-map-file

Syntax

-memory-map-file *filename*

Description

Generate a GNU ld linker script from the SEGGER Embedded Studio memory map file *filename*.

-memory-map-macros

Syntax

-memory-map-macros *macro*=*value*[;*macro*=*value*]

Description

Define SEGGER Embedded Studio macros to use when reading a memory map file.

-section-placement-file

Syntax

-section-placement-file *filename*

Description

Generate a GNU ld linker script from the SEGGER Embedded Studio section placement file *filename*. If this option is used, a memory map file should also be specified with the *-memory-map-file* option.

-section-placement-macros

Syntax

-section-placement-macros *macro*=*value*[;*macro*=*value*]

Description

Define SEGGER Embedded Studio macros to use when reading a section placement file.

-symbols

Syntax

-symbols *symbol*=*value*[;*symbol*=*value*]

Description

Add extra symbol definitions to the ld linker script.

Package generator

To create a package the program **mkpkg** can be used. The set of files to put into the package should be in the desired location in the `$(PackagesDir)` directory. The **mkpkg** command should be run with `$(PackagesDir)` as the working directory and all files to go into the package must be referred to using relative paths. A package must have a package description file that is placed in the `$(PackagesDir)/packages` directory. The package description file name must end with `_package.xml`. If a package is to create entries in the new project wizard then it must have a file name `project_templates.xml`.

For example, a package for the mythical FX150 processor would supply the following files:

A project template file called `targets/FX150/project_templates.xml`. The format of the project templates file is described in **Project Templates file format**.
The `$(PackagesDir)`-relative files that define the functionality of the package.
A package description file called `packages/FX150_package.xml`. The format of the package description file is described in **Package Description file format**.

The package file `FX150.emPackage` would be created using the following command line:

```
mkpkg -c packages/FX150.emPackage targets/FX150/project_templates.xml  packages/
FX150_package.xml
```

You can list the contents of the package using the **-t** option:

```
mkpkg -t packages/FX150.emPackage
```

You can remove an entry from a package using the **-d** option:

```
mkpkg -d packages/FX150.emPackage -d fileName
```

You can add or replace a file into an existing package using the **-r** option:

```
mkpkg -r packages/FX150.emPackage -r fileName
```

You can extract files from an existing package using the **-x** option:

```
mkpkg -x packages/FX150.emPackage outputDirectory
```

You can automate the package creation process using a **Combining** project type.

Using the new project wizard create a combining project in the directory `$(PackagesDir)`.
Set the **Output File Path** property to be `$(PackagesDir)/packages/mypackage.emPackage`.
Set the **Combine command** property to **$(StudioDir)/bin/mkpkg -c $(CombiningOutputFilePath) $(CombiningRelInputPaths)**.
Add the files you want to go into the package into the project using the Project Explorer.
Right-click the project node in the Project Explorer and choose **Build**.

When a package is installed, the files in the package are copied into the desired `$(PackagesDir)`-relative locations. When a file is copied into the `$(PackagesDir)/packages` directory and its filename ends with `_package.xml` the file `$(PackagesDir)/packages/installed_packages.xml` is updated with an entry:

```
<include filename="FX150_package.xml" />
```

During development of a package you can manually edit this file. The same applies to the file `$(PackagesDir)/targets/project_templates.xml` which will contain a reference to your `project_templates.xml` file.

Usage:

`mkpkg` [*options*] *packageFileName file1 file2*

Option	Description
`-c`	Create a new package.
`-compress level`	Change compression level (0 for none, 9 for maximum).
`-d`	Remove files from a package.
`-f`	Output files to stdout.
`-r`	Replace files in a package.
`-readonly`	Force all files to have read only attribute.
`-t`	List the contents of a package.
`-v`	Be chatty.
`-V`	Show version information.
`-x`	Extract files from a package.

Appendices

File formats

This section describes the file formats SEGGER Embedded Studio uses:

Memory Map file format

SEGGER Embedded Studio memory-map files are structured using XML syntax for its simple construction and parsing.

The first entry of the project file defines the XML document type used to validate the file format.

```
<!DOCTYPE Board_Memory_Definition_File>
```

The next entry is the `Root` element. There can only be one `Root` element in a memory map file:

```
<Root name="My Board">
```

A `Root` element has a `name` attribute every element in a memory map file has a `name` attribute. Names should be unique within a hierarchy level. Within a `Root` element, there are `MemorySegment` elements that represent regions within the memory map.

```
<Root name="My Board">
  <MemorySegment name="Flash" start="0x1000" size="0x200" access="ReadOnly">
```

`MemorySegment` elements have the following attributes:

> *start*:The start address of the memory segment. A simple expression, usually a hexadecimal number with a 0x prefix.
> *size*:The size of the memory segment. A simple expression, usually a hexadecimal number with a 0x prefix.
> *access*:The permissible access types of the memory segment. One of `ReadOnly`, `Read/Write`, `WriteOnly`, or `None`.
> *address_symbol*:A symbolic name for the start address of the memory segment.
> *size_symbol*:A symbolic name for the size of the memory segment.
> *address_symbol*:A symbolic name for the end address of the memory segment.

`RegisterGroup` elements are used to organize registers into groups. `Register` elements are used to define peripheral registers:

```
 <Root name="My Board" >
  <MemorySegment name="System" start="0x2000" size="0x200" >
    <RegisterGroup name="Peripheral1" start="0x2100" size="0x10" >
      <Register name="Register1" start="+0x8" size="4" >
```

`RegisterGroup` elements have the same attributes as `MemorySegment` elements. `Register` elements have the following attributes:

> *name*:Register names should be valid C/C++ identifier names, i.e., alphanumeric characters and underscores are allowed but names cannot start with a number.
> *start*:The start address of the memory segment. Either a C-style hexadecimal number or, if given a + prefix, an offset from the enclosing element's start address.
> *size*:The size of the register in bytes, either 1, 2, or 4.

access:The same as the `access` attribute of the `MemorySegment` element.

address_symbol:The same as the `address_symbol` attribute of the `MemorySegment` element.

A `Register` element can contain `BitField` elements that represent the bits in a peripheral register:

```
 <Root name="My Board" >
  <MemorySegment name="System" start="0x2000" size="0x200" >
    <RegisterGroup name="Peripheral1" start="0x2100" size="0x10" >
      <Register name="Register1" start="+0x8" size="4" >
        <BitField name="Bits_0_to_3" start="0" size="4" />
```

`BitField` elements have the following attributes:

name:The same as the `name` attribute of the `RegisterGroup` element.

start:The starting bit position, 031.

size:The total number of bits, 132.

A `Bitfield` element can contain `Enum` elements:

```
<Root name="My Board" >
  <RegisterGroup name="Peripheral1" start="0x2100" size="0x10" >
    <Register name="Register1" start="+0x8" size="4" >
      <BitField name="Bits_0_to_3" start="0" size="4" />
        <Enum name="Enum3" start="3" />
        <Enum name="Enum5" start="5" />
```

You can import CMSIS SVD files (see **http://www.onarm.com/**) into a memory map using the `ImportSVD` element:

```
<ImportSVD filename="$(TargetsDir)/targets/Manufacturer1/Processor1.svd.xml">
```

The `filename` attribute is an absolute filename which is macro-expanded using SEGGER Embedded Studio system macros.

When a memory map file is loaded either for the memory map viewer or to be used for linking or debugging, it is preprocessed using the (as yet undocumented) SEGGER Embedded Studio XML preprocessor.

Section Placement file format

SEGGER Embedded Studio section-placement files are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type used to validate the file format:

```
<!DOCTYPE Linker_Placement_File>
```

The next entry is the `Root` element. There can only be one `Root` element in a memory map file:

```
<Root name="Flash Placement">
```

A `Root` element has a `name` attribute. Every element in a section-placement file has a `name` attribute. Each name should be unique within its hierarchy level. Within a `Root` element, there are `MemorySegment` elements. These correspond to memory regions defined in a memory map file that will be used in conjunction with the section-placement file when linking a program. For example:

```
 <Root name="Flash Placement">
  <MemorySegment name="FLASH">
```

A `MemorySegment` contains `ProgramSection` elements that represent program sections created by the C/C++ compiler and assembler. The order of `ProgramSection` elements within a `MemorySegment` element represents the order in which the sections will be placed when linking a program. The first `ProgramSection` will be placed first and the last one will be placed last.

```
<Root name="My Board" >
  <MemorySegment name="FLASH">
    <ProgramSection name=".text">
```

`ProgramSection` elements have the following attributes:

address_symbol:A symbolic name for the start address of the section.
alignment:The required alignment of the program section; a decimal number specifying the byte alignment.
end_symbol:A symbolic name for the end address of the section.
fill:The optional value used to fill unspecified regions of memory, a hexadecimal number with a 0x prefix.
inputsections:An expression describing the input sections to be placed in this section. If you omit this (recommended) and the section name isn't one of **.text**, **.dtors**, **.ctors**, **.data**, **.rodata**, or **.bss**, then the equivalent input section of ***(.name .name.*)** is supplied to the linker.
keep:If **Yes**, the section will be kept even if none of the symbols are referenced by the rest of the program.
load:If **Yes**, the section is loaded. If **No**, the section isn't loaded.
place_from_segment_end:If **Yes**, this section and following sections will be placed at the end of the segment. Please note that this will only succeed if the section and all following sections have a fixed size specified with the **size** attribute.
runin:This specifies the name of the section to copy this section to.

runoffset:This specifies an offset from the load address that the section will be run from.

size:The optional size of the program section in bytes, a hexadecimal number with a 0x prefix.

size_symbol:A symbolic name for the size of the section.

start:The optional start address of the program section, a hexadecimal number with a 0x prefix.

When a section placement file is used for linking it is preprocessed using the (as yet undocumented) SEGGER Embedded Studio XML preprocessor.

Project file format

SEGGER Embedded Studio project files are held in text files with the `.emProject` extension. Because you may want to edit project files, and perhaps generate them, they are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type used to validate the file format:

```
<!DOCTYPE CrossStudio_Project_File>
```

The next entry is the `solution` element; there can only be one `solution` element in a project file. This specifies the solution name displayed in the **Project Explorer** and has a version attribute that defines the file-format version of the project file. Solutions can contain projects, projects can contain folders and files, and folders can contain folders and files. This hierarchy is reflected in the XML nestingfor example:

```
<solution version="1" Name="solutionname">
  <project Name="projectname">
    <file Name="filename" />
    <folder Name="foldername">
      <file Name="filename2" />
    </folder>
  </project>
</solution>
```

Note that each entry has a `Name` attribute. Names of `project` elements must be unique to the solution, and names of `folder` elements must be unique to the project, but names of files do not need to unique.

Each `file` element must have a `file_name` attribute that is unique to the project. Ideally, the `file_name` is a file path relative to the project (or solution directory), but you can also specify a full file path, if you want to. File paths are case-sensitive and use "/" as the directory separator. They may contain macro instantiations, so file paths cannot contain the "$" character. For example

```
<file file_name="$(StudioDir)/source/crt0.s" Name="crt0.s" />
```

will be expanded using the value of **$(StudioDir)** when the file is referenced from SEGGER Embedded Studio.

Project properties are held in configuration elements with the `Name` attribute of the configuration element corresponding to the configuration name, e.g., "Debug". At a given project level (i.e., solution, project, folder), there can only be one named configuration elementi.e., all properties defined for a configuration are in single configuration element.

```
<project Name="projectname">

  <configuration project_type="Library" Name="Common" />
  <configuration Name="Release" build_debug_information="No" />

</project>
```

You can use the `import` element to link projects:

```
<import file_name="target/libc.emProject" />
```

Project Templates file format

The SEGGER Embedded Studio **New Project** dialog works from a file called `project_templates.xml` in the `targets` subdirectory of the SEGGER Embedded Studio installation directory. Because you may want to add your own new project types, they are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type used to validate the file format:

```
<!DOCTYPE Project_Templates_File>
```

The next entry is the `projects` element, which is used to group a set of new project entries into an XML hierarchy.

```
<projects>
  <project>
</projects>
```

Each entry has a `project` element that contains the class of the project (attribute `caption`), the name of the project (attribute `name`), its type (attribute `type`) and a description (attribute `description`). For example:

```
<project caption="ARM Evaluator7T" name="Executable"
  description="An executable for an ARM Evaluator7T." type="Executable"/>
```

The project type can be one of these:

Executable: a fully linked executable.

Library: a static library.

Object file: an object file.

Staging: a staging project.

Combining: a combining project.

Externally Built Executable: an externally built executable.

The configurations to be created for the project are defined using the `configuration` element, which must have a `name` attribute:

```
<configuration name="ARM RAM Release"/>
```

The property values to be created for the project are defined using the `property` element. If you have a defined value, you can specify this using the `value` attribute and, optionally, set the property in a defined `configuration`, such as:

```
<property name="target_reset_script" configuration="RAM"
  value="Evaluator7T_ResetWithRamAtZero()" />
```

Alternatively, you can include a property that will be shown to the user, prompting them to supply a value as part of the new-project process.

```
<property name="linker_output_format"/>
```

The folders to be created are defined using the `folder` element. The `folder` element must have a `name` attribute and can also have a `filter` attribute. For example:

```
<folder name="Source Files" filter="c;cpp;cxx;cc;h;s;asm;inc" />
```

The files to be in the project are specified using the `file` element. You can use build-system macros (see **Project macros**) to specify files located in the SEGGER Embedded Studio installation directory. Files will be copied to the project directory or just left as references, depending on the value of the `expand` attribute:

```
<file name="$(StudioDir)/source/crt0.s" expand="no"/>
```

You can define the set of configurations that can be referred to in the top-level `configurations` element:

```
<configurations>
  <configuration>
</configurations>
```

This contains the set of all configurations that can be created when a project is created. Each configuration is defined using a `configuration` element, which can define the property values for that configuration. For example:

```
<configuration name="Debug">
  <property name="build_debug_information" value="Yes">
```

Property Groups file format

The SEGGER Embedded Studio project system provides a means to create new properties that change a number of project property settings and can also set C pre-processor definitions when selected. Such properties are called *property groups* and are defined in a property-groups file. The property-group file to use for a project is defined by the **Property Groups File** property. These files usually define target-specific properties and are structured using XML syntax to enable simple construction and parsing.

The first entry of the property groups file defines the XML document type, which is used to validate the file format:

```
<!DOCTYPE CrossStudio_Group_Values>
```

The next entry is the `propertyGroups` element, which is used to group a set of property groups entries into an XML hierarchy:

```
<propertyGroups>
  <grouphdots

  <grouphdots
</propertyGroups>
```

Each group has the name of the group (attribute `name`), the name of the options category (attribute `group`), short (attribute `short`) and long (attribute `long`) help descriptions, and a default value (attribute `default`). For example:

```
<group short="Target Processor" group="Build Options" short="Target Processor"
  long="Select a set of target options" name="Target" default="STR912FW44" />
```

Each group has a number of `groupEntry` elements that define the enumerations of the group.

```
<group\>
  <groupEntry>

  <groupEntry>
</group>
```

Each `groupEntry` has the name of the entry (attribute `name`), e.g.:

```
<groupEntry name="STR910FW32">
```

A `groupEntry` has the property values and C pre-processor definitions that are set when the `groupEntry` is selected; they are specified with `property` and `cdefine` elements. For example:

```
<groupEntry>
  <property>
  <cdefine>
  <property>
</groupEntry>
```

A `property` element has the property's name (attribute `name`), its value (attribute `value`), and an optional configuration (attribute `configuration`):

```
<property name="linker_memory_map_file"
  value="$(StudioDir)/targets/ST_STR91x/ST_STR910FM32_MemoryMap.xml" />
```

A `cdefine` element has the C preprocessor name (attribute `name`) and its value (attribute `value`):

```
<cdefine value="STR910FM32" name="TARGET_PROCESSOR" />
```

Package Description file format

Package-description files are XML files used by SEGGER Embedded Studio to describe a support package, its contents, and any dependencies it has on other packages.

Each package file must contain one `package` element that describes the package. Optionally, the `package` element can contain a collection of `file`, `history`, and `documentation` elements to be used by SEGGER Embedded Studio for documentation purposes.

The filename of the package-description file should match that of the package and end in "*_package.xml*".

Below is an example of two package-description files. The first is for a base chip-support package for the LPC2000; the second is for a board-support package dependent on the first:

Philips_LPC2000_package.xml

```
<!DOCTYPE CrossStudio_Package_Description_File>
<package cpu_manufacturer="Philips" cpu_family="LPC2000" version="1.1" ses_versions="8:1-"
 author="SEGGER" >
  <file file_name="$(TargetsDir)/Philips_LPC210X/arm_target_Philips_LPC210X.htm"
 title="LPC2000 Support Package Documentation" />
  <file file_name="$(TargetsDir)/Philips_LPC210X/Loader.emProject" title="LPC2000 Loader
 Application Solution" />
  <group title="System Files">
    <file file_name="$(TargetsDir)/Philips_LPC210X/Philips_LPC210X_Startup.s" title="LPC2000
 Startup Code" />
    <file file_name="$(TargetsDir)/Philips_LPC210X/Philips_LPC210X_Target.js" title="LPC2000
 Target Script" />
  </group>
  <history>
    <version name="1.1" >
      <description>Corrected LPC21xx header files and memory maps to include GPIO ports 2
 and 3.</description>
      <description>Modified loader memory map so that .libmem sections will be placed
 correctly.</description>
    </version>
    <version name="1.0" >
      <description>Initial Release.</description>
    </version>
  </history>
  <documentation>
    <section name="Supported Targets">
      <p>This CPU support package supports the following LPC2000 targets:
        <ul>
          <li>LPC2103</li>
          <li>LPC2104</li>
          <li>LPC2105</li>
          <li>LPC2106</li>
          <li>LPC2131</li>
          <li>LPC2132</li>
          <li>LPC2134</li>
          <li>LPC2136</li>
          <li>LPC2138</li>
        </ul>
      </p>
    </section>
```

```
  </documentation>
</package>
```

CrossFire_LPC2138_package.xml

```
<!DOCTYPE CrossStudio_Package_Description_File>
<package cpu_manufacturer="Philips" cpu_family="LPC2000" cpu_name="LPC2138"
 board_manufacturer="Rowley Associates" board_name="CrossFire LPC2138"
 dependencies="Philips_LPC2000" version="1.0">
  <file file_name="$(SamplesDir)/CrossFire_LPC2138/CrossFire_LPC2138.emProject"
 title="CrossFire LPC2138 Samples Solution" />
  <file file_name="$(SamplesDir)/CrossFire_LPC2138/ctl/ctl.emProject" title="CrossFire
 LPC2138 CTL Samples Solution" />
</package>
```

Package elements

The `package` element describes the support package, its contents, and any dependencies it has on other packages. Valid attributes for this element are:

Attribute	Description
`author`	The author of the package.
`board_manufacturer`	The manufacturer of the board supported by the package *(if omitted, CPU manufacturer will be used)*.
`board_name`	The name of the specific board supported by the package *(only required for board-support packages)*.
`cpu_family`	The family name of the CPU supported by the package *(optional)*.
`cpu_manufacturer`	The manufacturer of the CPU supported by the package.
`cpu_name`	The name of the specific CPU supported by the package *(may be omitted if the CPU family is specified)*.
`ses_versions`	A string describing which version of SEGGER Embedded Studio supports the package *(optional)*. The format of the string is *target_id_number:version_range_string*.
`description`	A description of the package *(optional)*.
`dependencies`	A semicolon-separated list of packages the package requires to be installed in order to work.
`installation_directory`	The directory in which the package should be installed *(optional\--if undefined, defaults to "$(PackagesDir)")*.
`title`	A short description of the package *(optional)*.

`version`	The package version number.

File elements

The `file` element is used by SEGGER Embedded Studio for documentation purposes by adding links to files of interest within the package such as example project files and documentation.

Attribute	Description
`file_name`	The file path of the file.
`title`	A description of the file.

Optionally, `file` elements can be grouped into categories using the `group` element.

Group elements

The `group` element is used for categorizing files described by `file` elements into a particular group.

Attribute	Description
`title`	Title of the group.

History elements

The `history` element is used to hold a description of the package's version history.

The `history` element should contain a collection of `version` elements.

Version element

The `version` element is used to hold the description of a particular version of the package.

Attribute	Description
`name`	The name of the version being described.

The `version` element should contain a collection of `description` elements.

Description elements

Each `description` element contains text that describes a feature of the package version.

Documentation elements

The `documentation` element is used to provide arbitrary documentation for the package.

The `documentation` element should contain a collection of one or more `section` elements.

Section elements

The `section` element contains package documentation in XHTML format.

Attribute	Description
`name`	The title of the documentation section.

target_id_number

The following table lists the possible target ID numbers:

Target	ID
AVR	4
ARM	8
MSP430	9
MAXQ20	18
MAXQ30	19

version_range_string

The `version_range_string` can be any of the following:

version_number:The package will only work on *version_number*.
version_number-:The package will work on *version_number* or any future version.
-version_number:The package will work on *version_number* or any earlier version.
low_version_number-high_version_number:The package will work on *low_version_number*, *high_version_number* or any version in between.

External Tools file format

SEGGER Embedded Studio external-tool configuration files are structured using XML syntax for its simple construction and parsing.

Tool configuration files

The SEGGER Embedded Studio application will read the tool configuration file when it starts up. By default, SEGGER Embedded Studio will read the file **$(StudioUserDir)/tools.xml**.

Structure

All tools are wrapped in a **tools** element:

```
<tools>

</tools>
```

Inside the tools element are **item** elements that define each tool:

```
<tools>
  <item name="logical name">

  </item>
</tools>
```

The **item** element requires an **name** attribute, which is an internal name for the tool, and has an optional *wait* element. When SEGGER Embedded Studio invokes the tool on a file or project, it uses the *wait* element to determine whether it should wait for the external tool to complete before continuing. If the *wait* attribute is not provided or is set to *yes*, SEGGER Embedded Studio will wait for external tool to complete.

The way that the tool is presented in SEGGER Embedded Studio is configured by elements inside the

element.

menu

The **menu** element defines the wording used inside menus. You can place a shortcut to the menu using an ampersand, which must be escaped using **&amp** in XML, before the shortcut letter. For instance:

```
<menu>&amp;PC-lint (Unit Check)</menu>
```

text

The optional **text** element defines the wording used in contexts other than menus, for instance when the tool appears as a tool button with a label. If **text** is not provided, the tool's textual appearance outside the menu is taken from the **menu** element (and is presented without an shortcut underline). For instance:

```
<text>PC-lint (Unit Check)</text>
```

tip

The optional **tip** element defines the status tip, shown on the status line, when moving over the tool inside SEGGER Embedded Studio:

```
<tip>Run a PC-lint unit checkout on the selected file or folder</tip>
```

key

The optional **key** element defines the accelerator key, or key chord, to use to invoke the tool using the keyboard. You can construct the key sequence using modifiers **Ctrl**, **Shift**, and **Alt**, and can specify more than one key in a sequence (note: Windows and Linux only; OS X does not provide key chords). For instance:

```
<key>Ctrl+L, Ctrl+I</key>
```

message

The optional **message** element defines the text shown in the tool log in SEGGER Embedded Studio when running the tool. For example:

```
<message>Linting</message>
```

match

The optional **match** element defines which documents the tool will operator on. The match is performed using the file extension of the document. If the file extension of the document matches one of the wildcards provided, the tool will run on that document. If there is no **match** element, the tool will run on all documents. For instance:

```
<match>*.c;*.cpp</match>
```

commands

The **commands** element defines the command line to run to invoke the tool. The command line is expanded using macros applicable to the file derived from the current build configuration and the project settings. Most importantly, the standard **$(InputPath)** macro expands to a full pathname for the target file.

Additional macros constructed by SEGGER Embedded Studio are:

$(DEFINES) is the set of **-D** options applicable to the current file, derived from the current configuration and project settings.
$(INCLUDES) is the set of **-I** options applicable to the current file, derived from the current configuration and project settings.

For instance:

```
<commands>
  &quot;$(LINTDIR)/lint-nt&quot; -i$(LINTDIR)/lnt &quot;$(LINTDIR)/lnt/co-gcc.lnt&quot;
  $(DEFINES) $(INCLUDES) -D__GNUC__ -u -b +macros -w2 -e537 +fie +ffn -width(0,4) -hF1
  &quot;-format=%f:%l:%C:s%t:s%m&quot; &quot;$(InputPath)&quot;
</commands>
```

In this example we intend **$(LINTDIR)** to point to the directly where PC-lint is installed and for **$(LINTDIR)** to be defined as a SEGGER Embedded Studio global macro. You can set global macros using **ide_environment_options_dialog**.

Note that additional **&quot** entities are placed around pathnames in the **commands** sectionthis is to ensure that paths that contain spaces are correctly interpreted when the command is executed by SEGGER Embedded Studio.

Building Environment Options

Build

Property	Description
Automatically Build Before Debug `Environment/Build/Build Before Debug`Boolean	Enables auto-building of a project before downloading if it is out of date.
Build Macros `Environment/Macros/Global Macros`StringList	Build macros that are shared across all solutions and projects e.g. paths to library files.
Confirm Debugger Stop `Environment/Build/Confirm Debugger Stop`Boolean	Present a warning when you start to build that requires the debugger to stop.
Display ETA `Environment/Build/Display ETA`Boolean	Selects whether to attempt to compute and display the ETA on building.
Display Progress Bar `Environment/Build/Display Progress Bar`Boolean	Selects whether to display progress bar on building.
Echo Build Command Lines `Environment/Build/Show Command Lines`Boolean	Selects whether build command lines are written to the build log.
Echo Raw Error/Warning Output `Environment/Build/Show Unparsed Error Output`Boolean	Selects whether the unprocessed error and warning output from tools is displayed in the build log.
Find Error After Building `Environment/Build/Find Error After Build`Boolean	Moves the cursor to the first diagnostic after a build completes with errors.
Keep Going On Error `Environment/Build/Keep Going On Error`Boolean	Build doesn't stop on error.
Save Project File Before Building `Environment/Build/Save Project File On Build`Boolean	Selects whether to save the project file prior to build.
Show Build Information `Environment/Build/Show Build Information`Boolean	Show build information.
Toolchain Root Directory `Environment/Build/Tool Chain Root Directory`String	Specifies where to find the toolchain (compilers etc).

Build Acceleration

Property	Description
Disable Unity Build `Environment/Build/Disable Unity Build`Boolean	Ignore Unity Build project properties and always build individual project components.
Parallel Building Threads `Environment/Build/Building Threads`IntegerRange	The number of threads to launch when building.

Compatibility

Property	Description
Compiler Supports Section Renaming `ARM/Build/Compiler Can Rename Sections`Boolean	Compiler supports the -mtext=t, -mdata=d, -mbss=b, -mrodata=r section renaming options.
Default Assembler Variant `ARM/Build/Assembler Variant Default`Enumeration	Specifies the default assembler variant to use.
Default Compiler Variant `ARM/Build/Compiler Variant Default`Enumeration	Specifies the default linker variant to use.
Installation Directory `ARM/Build/StudioDir Directory`DirPath	The installation directory to be used for building - the value $(StudioDir) is set to.
Use External GCC `ARM/Build/Use External GCC`Boolean	Use an external GCC toolchain for the build.

Window

Property	Description
Show Build Log On Build `Environment/Show Transcript On Build`Boolean	Show the build log when a build starts.

Debugging Environment Options

Breakpoint

Property	Description
Clear Disassembly Breakpoints On Debug Stop `Environment/Debugger/Clear Disassembly Breakpoint`Boolean	Clear Disassembly Breakpoints On Debug Stop

Display

Property	Description
Close Disassembly On Mode Switch `Environment/Debugger/Close Disassembly On Mode Switch`Boolean	Close Disassembly On Mode Switch
Data Tips Display a Maximum Of `Environment/Debugger/Maximum Array Elements Displayed`IntegerRange	Selects the maximum number of array elements displayed in a datatip.
Default Display Mode `Environment/Debugger/Default Variable Display Mode`Enumeration	Selects the format that data values are shown in.
Display Floating Point Number In `Environment/Debugger/Floating Point Format Display`Custom	The printf format directive used to display floating point numbers.
Maximum Backtrace Calls `Environment/Debugger/Maximum Backtrace Calls`IntegerRange	Selects the maximum number of calls when backtracing.
Prompt To Display If More Than `Environment/Debugger/Array Elements Prompt Size`IntegerRange	The array size to display with prompt.
Show Data Tips In Text Editor `Environment/Debugger/Show Data Tips`Boolean	Show Data Tips In Text Editor
Show Labels In Disassembly `Environment/Debugger/Disassembly Show Labels`Boolean	Show Labels In Disassembly
Show Source In Disassembly `Environment/Debugger/Disassembly Show Source`Boolean	Show Source In Disassembly
Show char * As Null Terminated String `Environment/Debugger/Display Char Ptr As String`Boolean	Display char * as null terminated string.

Source Path `Environment/Debugger/Source Path`StringList	Global search path to find source files.

Extended Data Tips

Property	Description
ASCII `Environment/Debugger/Extended Tooltip Display Mode/ASCII`Boolean	Selects ASCII extended datatips.
Binary `Environment/Debugger/Extended Tooltip Display Mode/Binary`Boolean	Selects Binary extended datatips.
Decimal `Environment/Debugger/Extended Tooltip Display Mode/Decimal`Boolean	Selects Decimal extended datatips.
Hexadecimal `Environment/Debugger/Extended Tooltip Display Mode/Hexadecimal`Boolean	Selects Hexadecimal extended datatips.
Octal `Environment/Debugger/Extended Tooltip Display Mode/Octal`Boolean	Selects Octal extended datatips.
Unsigned Decimal `Environment/Debugger/Extended Tooltip Display Mode/Unsigned Decimal`Boolean	Selects Unsigned Decimal extended datatips.

Window

Property	Description
Clear Debug Terminal On Run `Environment/Clear Debug Terminal On Run`Boolean	Clear the debug terminal automatically when a program is run.
Hide Output Window On Successful Load `Debugging/Hide Transcript On Successful Load`Boolean	Hide the Output window when a load completes without error.
Show Target Log On Load `Debugging/Show Transcript On Load`Boolean	Show the target log when a load starts.

IDE Environment Options

Browser

Property	Description
Text Size `Environment/Browser/Text Size`Enumeration	Sets the text size of the integrated HTML and help browser.
Underline Hyperlinks In Browser `Environment/Browser/Underline Web Links`Boolean	Enables underlining of hypertext links in the integrated HTML and help browser.

File Extension

Property	Description
ELF Executable File Extensions `ElfDwarf/Environment/Executable File Extensions`StringList	The file extensions used for ELF executable files.
ELF Object File Extensions `ElfDwarf/Environment/Object File Extensions`StringList	The file extensions used for ELF object files.

File Search

Property	Description
Files To Search `Find In Files/File Type`StringList	The wildcard used to match files in Find In Files searches.
Find History `Find In Files/Find History`StringList	The list of strings recently used in searches.
Folder History `Find In Files/Folder History`StringList	The set of folders recently used in file searches.
Match Case `Find In Files/Match Case`Boolean	Whether the case of letters must match exactly when searching.
Match Whole Word `Find In Files/Match Whole Word`Boolean	Whether the whole word must match when searching.
Replace History `Find In Files/Replace History`StringList	The list of strings recently used in searches.

Search Dependencies `Find In Files/Search Dependencies`Boolean	Controls searching of dependent files.
Search In `Find In Files/Context`Enumeration	Where to look to find files.
Use Regular Expressions `Find In Files/Use RegExp`Boolean	Whether to use a regular expression or plain text search.

Internet

Property	Description
Automatically Check For Packages `Environment/Internet/Check Packages`Boolean	Specifies whether to enable downloading of the list of available packages.
Automatically Check For Updates `Environment/Internet/Check Updates`Boolean	Specifies whether to enable checking for software updates.
Check For Latest News `Environment/Internet/RSS Update`Boolean	Specifies whether to enable downloading of the Latest News RSS feeds.
Enable Connection Debugging `Environment/Internet/Enable Debugging`Boolean	Controls debugging traces of internet connections and downloads.
External Web Browser `Environment/External Web Browser`FileName	The path to the external web browser to use when accessing non-local files.
HTTP Proxy Host `Environment/Internet/HTTP Proxy Server`String	Specifies the IP address or hostname of the HTTP proxy server. If empty, no HTTP proxy server will be used.
HTTP Proxy Port `Environment/Internet/HTTP Proxy Port`IntegerRange	Specifies the HTTP proxy server's port number.
Maximum Download History Items `Environment/Internet/Max Download History Items`IntegerRange	The maximum amount of download history kept in the downloads window.
Use Content Delivery Network `Environment/Package/Use Content Delivery Network`Boolean	Specifies whether to use content delivery network to deliver packages.

Package Manager

Property	Description
Check Solution Package Dependencies `Environment/Package/Check Solution Package Dependencies`Boolean	Specifies whether to check package dependencies when a solution is loaded.

Package Directory `Environment/Package/Destination Directory`String	Specifies the directory packages are installed to.
Show Check For Packages Dialog `Environment/Package/Show Check For Packages Dialog`Boolean	Specifies whether the package manager should prompt for a package list refresh.
Show Logos `Environment/Package/Show Logos`Enumeration	Specifies whether the package manager should display company logos.

Print

Property	Description
Bottom Margin `Environment/Printing/Bottom Margin`IntegerRange	The page's bottom margin in millimetres.
Left Margin `Environment/Printing/Left Margin`IntegerRange	The page's left margin in millimetres.
Page Orientation `Environment/Printing/Orientation`Enumeration	The page's orientation.
Page Size `Environment/Printing/Page Size`Enumeration	The page's size.
Right Margin `Environment/Printing/Right Margin`IntegerRange	The page's right margin in millimetres.
Top Margin `Environment/Printing/Top Margin`IntegerRange	The page's top margin in millimetres.

Startup

Property	Description
Allow Multiple SEGGER Embedded Studios `Environment/Permit Multiple Studio Instances`Boolean	Allow more than one SEGGER Embedded Studio to run at the same time.
Load Last Project On Startup `Environment/Load Last Project On Startup`Boolean	Specifies whether to load the last project the next time SEGGER Embedded Studio runs.
New Project Directory `Environment/General/Solution Directory`String	The directory where projects are created.
Splash Screen `Environment/Splash Screen`Enumeration	How to display the splash screen on startup.

Status Bar

Property	Description
(Visible) `Environment/Status Bar`Boolean	Show or hide the status bar.
Show Build Status Pane `Environment/General/Status Bar/Show Build Status`Boolean	Show or hide the Build pane in the status bar.
Show Caret Position Pane `Environment/General/Status Bar/Show Caret Pos`Boolean	Show or hide the Caret Position pane in the status bar.
Show Insert/Overwrite Status Pane `Environment/General/Status Bar/Show Insert Mode`Boolean	Show or hide the Insert/Overwrite pane in the status bar.
Show Read-Only Status Pane `Environment/General/Status Bar/Show Read Only`Boolean	Show or hide the Read Only pane in the status bar.
Show Size Grip `Environment/General/Status Bar/Show Size Grip`Boolean	Show or hide the status bar size grip.
Show Target Pane `Environment/General/Status Bar/Show Target`Boolean	Show or hide the Target pane in the status bar.
Show Time Pane `Environment/General/Status Bar/Show Time`Boolean	Show or hide the Time pane in the status bar.

User Interface

Property	Description
Application Main Font `Environment/Application Main Font`Font	The font to use for the user interface as a whole.
Application Monospace Font `Environment/Application Monospace Font`FixedPitchFont	The fixed-size font to use for the user interface as a whole.
Error Display Timeout `Environment/Error Display Timeout`IntegerRange	The minimum time, in seconds, that errors are shown for in the status bar.
Errors Are Displayed `Environment/Error Display Mode`Enumeration	How errors are reported in SEGGER Embedded Studio.

File Size Display Units `Environment/Size Display Unit`Enumeration	How to display sizes of items in the user interface. SI defines 1kB=1000 bytes, IEC defines 1kiB=1024 bytes, Alternate SI defines 1kB=1024 bytes.
Number File Names in Menus `Environment/Number Menus`Boolean	Number the first nine file names in menus for quick keyboard access.
Show Large Icons In Toolbars `Environment/General/Large Icons`Boolean	Show large or small icons on toolbars.
Show Ribbon `Environment/General/Ribbon/Show`Boolean	Show or hide the ribbon.
Show Window Selector On Ctrl+Tab `Environment/Show Selector`Boolean	Present the Window Selector on Next Window and Previous Window commands activated from the keyboard.
User Interface Theme `Environment/General/Skin`Enumeration	The theme that SEGGER Embedded Studio uses.
Window Menu Contains At Most `Environment/Max Window Menu Items`IntegerRange	The maximum number of windows appearing in the Windows menu.

Programming Language Environment Options

Assembly Language

Property	Description
Column Guide Columns `Text Editor/Indent/Assembly Language/Column Guides`String	The columns that guides are drawn for.
Indent Closing Brace `Text Editor/Indent/Assembly Language/Close Brace`Boolean	Indent the closing brace of compound statements.
Indent Context `Text Editor/Indent/Assembly Language/Context Lines`IntegerRange	The number of lines to use for context when indenting.
Indent Mode `Text Editor/Indent/Assembly Language/Indent Mode`Enumeration	How to indent when a new line is inserted.
Indent Opening Brace `Text Editor/Indent/Assembly Language/Open Brace`Boolean	Indent the opening brace of compound statements.
Indent Size `Text Editor/Indent/Assembly Language/Size`IntegerRange	The number of columns to indent a code block.
Tab Size `Text Editor/Indent/Assembly Language/Tab Size`IntegerRange	The number of columns between tabstops.
Use Tabs `Text Editor/Indent/Assembly Language/Use Tabs`Boolean	Insert tabs when indenting.
User-Defined Keywords `Text Editor/Indent/Assembly Language/Keywords`StringList	Additional identifiers to highlight as keywords.

C and C++

Property	Description
Column Guide Columns `Text Editor/Indent/C and C++/Column Guides`String	The columns that guides are drawn for.

Indent Closing Brace `Text Editor/Indent/C and C++/Close Brace`Boolean	Indent the closing brace of compound statements.
Indent Context `Text Editor/Indent/C and C++/Context Lines`IntegerRange	The number of lines to use for context when indenting.
Indent Mode `Text Editor/Indent/C and C++/Indent Mode`Enumeration	How to indent when a new line is inserted.
Indent Opening Brace `Text Editor/Indent/C and C++/Open Brace`Boolean	Indent the opening brace of compound statements.
Indent Size `Text Editor/Indent/C and C++/Size`IntegerRange	The number of columns to indent a code block.
Tab Size `Text Editor/Indent/C and C++/Tab Size`IntegerRange	The number of columns between tabstops.
Use Tabs `Text Editor/Indent/C and C++/Use Tabs`Boolean	Insert tabs when indenting.
User-Defined Keywords `Text Editor/Indent/C and C++/Keywords`StringList	Additional identifiers to highlight as keywords.

Default

Property	Description
Column Guide Columns `Text Editor/Indent/Default/Column Guides`String	The columns that guides are drawn for.
Indent Closing Brace `Text Editor/Indent/Default/Close Brace`Boolean	Indent the closing brace of compound statements.
Indent Context `Text Editor/Indent/Default/Context Lines`IntegerRange	The number of lines to use for context when indenting.
Indent Mode `Text Editor/Indent/Default/Indent Mode`Enumeration	How to indent when a new line is inserted.

Indent Opening Brace `Text Editor/Indent/Default/Open Brace`Boolean	Indent the opening brace of compound statements.
Indent Size `Text Editor/Indent/Default/Size`IntegerRange	The number of columns to indent a code block.
Tab Size `Text Editor/Indent/Default/Tab Size`IntegerRange	The number of columns between tabstops.
Use Tabs `Text Editor/Indent/Default/Use Tabs`Boolean	Insert tabs when indenting.
User-Defined Keywords `Text Editor/Indent/Default/Keywords`StringList	Additional identifiers to highlight as keywords.

Java

Property	Description
Column Guide Columns `Text Editor/Indent/Java/Column Guides`String	The columns that guides are drawn for.
Indent Closing Brace `Text Editor/Indent/Java/Close Brace`Boolean	Indent the closing brace of compound statements.
Indent Context `Text Editor/Indent/Java/Context Lines`IntegerRange	The number of lines to use for context when indenting.
Indent Mode `Text Editor/Indent/Java/Indent Mode`Enumeration	How to indent when a new line is inserted.
Indent Opening Brace `Text Editor/Indent/Java/Open Brace`Boolean	Indent the opening brace of compound statements.
Indent Size `Text Editor/Indent/Java/Size`IntegerRange	The number of columns to indent a code block.
Tab Size `Text Editor/Indent/Java/Tab Size`IntegerRange	The number of columns between tabstops.
Use Tabs `Text Editor/Indent/Java/Use Tabs`Boolean	Insert tabs when indenting.
User-Defined Keywords `Text Editor/Indent/Java/Keywords`StringList	Additional identifiers to highlight as keywords.

Source Control Environment Options

External Tools

Property	Description
Diff Command Line `Environment/Source Code Control/DiffCommand` StringList	The diff command line
Merge Command Line `Environment/Source Code Control/MergeCommand` StringList	The merge command line

Preference

Property	Description
Add Immediately `Environment/Source Code Control/ImmediateAdd` Boolean	Bypasses the confirmation dialog and immediately adds items to source control.
Commit Immediately `Environment/Source Code Control/ImmediateCommit` Boolean	Bypasses the confirmation dialog and immediately commits items.
Lock Immediately `Environment/Source Code Control/ImmediateLock` Boolean	Bypasses the confirmation dialog and immediately locks items.
Remove Immediately `Environment/Source Code Control/ImmediateRemove` Boolean	Bypasses the confirmation dialog and immediately removes items source control.
Resolved Immediately `Environment/Source Code Control/ImmediateResolved` Boolean	Bypasses the confirmation dialog and immediately mark items resolved.
Revert Immediately `Environment/Source Code Control/ImmediateRevert` Boolean	Bypasses the confirmation dialog and immediately revert items.
Unlock Immediately `Environment/Source Code Control/ImmediateUnlock` Boolean	Bypasses the confirmation dialog and immediately unlocks items.
Update Immediately `Environment/Source Code Control/ImmediateUpdate` Boolean	Bypasses the confirmation dialog and immediately updates items.

Text Editor Environment Options

Auto Recovery

Property	Description
Auto Recovery Backup Time `Text Editor/Auto Recovery Backup Time`IntegerRange	The time in minutes between saving of auto recovery backups files or 0 to disable generation of backup files.
Auto Recovery Keep Time `Text Editor/Auto Recovery Keep Time`IntegerRange	The time in days to keep unrecovered backup files or 0 to disable deletion of unrecovered backup files.

Cursor Fence

Property	Description
Bottom Margin `Text Editor/Margins/Bottom`IntegerRange	The number of lines in the bottom margin.
Keep Cursor Within Fence `Text Editor/Margins/Enabled`Boolean	Enable margins to fence and scroll around the cursor.
Left Margin `Text Editor/Margins/Left`IntegerRange	The number of characters in the left margin.
Right Margin `Text Editor/Margins/Right`IntegerRange	The number of characters in the right margin.
Top Margin `Text Editor/Margins/Top`IntegerRange	The number of lines in the right margin.

Editing

Property	Description
Allow Drag and Drop Editing `Text Editor/Drag Drop Editing`Boolean	Enables dragging and dropping of selections in the text editor.
Bold Popup Diagnostic Messages `Text Editor/Bold Popup Diagnostics`Boolean	Displays popup diagnostic messages in bold for easier reading.
Column-mode Tab `Text Editor/Column Mode Tab`Boolean	Tab key moves to the next textual column using the line above.
Confirm Modified File Reload `Text Editor/Confirm Modified File Reload`Boolean	Display a confirmation prompt before reloading a file that has been modified on disk.

Copy Action When Nothing Selected `Text Editor/Copy Action`Enumeration	What Copy copies when nothing is selected.
Cut Action When Nothing Selected `Text Editor/Cut Action`Enumeration	What Cut cuts when nothing is selected.
Cut Single Blank Line `Text Editor/Cut Blank Lines`Boolean	Selects whether to place text on the clipboard when a single blank line is cut. When set to
Diagnostic Cycle Mode `Text Editor/Diagnostic Cycle Mode`Enumeration	Iterates through diagnostics either from most severe to least severe or in reported order.
Edit Read-Only Files `Text Editor/Edit Read Only`Boolean	Allow editing of read-only files.
Enable Virtual Space `Text Editor/Enable Virtual Space`Boolean	Permit the cursor to move into locations that do not currently contain text.
Numeric Keypad Editing `Text Editor/Numeric Keypad Enabled`Boolean	Selects whether the numeric keypad plus and minus buttons copy and cut text.
Undo And Redo Behavior `Text Editor/Undo Mode`Enumeration	How Undo and Redo group your typing when it is undone and redone.

Find And Replace

Property	**Description**
Case Sensitive Matching `Text Editor/Find/Match Case`Boolean	Enables or disables the case sensitivity of letters when searching.
Find History `Text Editor/Find/History`StringList	The list of strings recently used in searches.
Regular Expression Matching `Text Editor/Find/Use RegExp`Boolean	Enables regular expression matching rather than plain text matching.
Replace History `Text Editor/Replace/History`StringList	The list of strings recently used in replaces.
Whole Word Matching `Text Editor/Find/Match Whole Word`Boolean	Enables or disables whole word matching when searching.

Formatting

Property	**Description**
Access Modifier Offset `Text Editor/Formatting/AccessModifierOffset`Integer	The extra indent or outdent of access modifiers, e.g. public:.

Align After Open Bracket `Text Editor/Formatting/` `AlignAfterOpenBracket`Boolean	If enabled, horizontally aligns arguments after an open bracket.
Align Escaped Newlines Left `Text Editor/Formatting/` `AlignEscapedNewlinesLeft`Boolean	If enabled, aligns escaped newlines as far left as possible otherwise puts them into the right-most column.
Align Operands `Text Editor/Formatting/` `AlignOperands`Boolean	If enabled, horizontally align operands of binary and ternary expressions.
Align Trailing Comments `Text Editor/Formatting/` `AlignTrailingComments`Boolean	If enabled, aligns trailing comments.

Allow All Parameters Of Declaration On Next Line
`Text Editor/Formatting/`
`AllowAllParametersOfDeclarationOnNextLineB`
	Allow putting all parameters of a function declaration onto the next line even if Bin-pack Parameters is disabled.

Allow Short 'if' Statements On A Single Line `Text Editor/Formatting/` `AllowShortIfStatementsOnASingleLine`Boolean	If enabled, short 'if' statements are put on a single line.
Allow Short Blocks On A Single Line `Text Editor/Formatting/` `AllowShortBlocksOnASingleLine`Boolean	If enabled, allows contracting simple braced statements to a single line.
Allow Short Case Labels On A Single Line `Text Editor/Formatting/` `AllowShortCaseLabelsOnASingleLine`Boolean	If enabled, short case labels will be contracted to a single line.
Allow Short Functions On A Single Line `Text Editor/Formatting/` `AllowShortFunctionsOnASingleLine`Enumeration	Optionally compress small functions to a single line.
Allow Short Loop Statements On A Single Line `Text Editor/Formatting/` `AllowShortLoopsOnASingleLine`Boolean	If enabled, short loop statements are put on a single line.
Always Break Before Multiline Strings `Text Editor/Formatting/` `AlwaysBreakAfterDefinitionReturnType`Boolean	If enabled, always break after function definition return types.
Always Break Before Multiline Strings `Text Editor/Formatting/` `AlwaysBreakBeforeMultilineStrings`Boolean	If enabled, always break before multiline strings.
Always Break Template Declarations `Text Editor/Formatting/` `AlwaysBreakTemplateDeclarations`Boolean	If enabled, always break after the 'template<...>' of a template declaration.
Bin-Pack Arguments `Text Editor/Formatting/` `BinPackArguments`Boolean	If disabled, a function call?s arguments will either be all on the same line or will have one line each.

Bin-Pack Parameters `Text Editor/Formatting/` `BinPackParameters`Boolean	If disabled, a function call's or function definition's parameters will either all be on the same line or will have one line each.
Break Before Binary Operators `Text Editor/Formatting/` `BreakBeforeBinaryOperators`Boolean	The way to wrap binary operators.
Break Before Braces `Text Editor/Formatting/` `BreakBeforeBraces`Enumeration	The brace breaking style to use.
Break Before Ternary Operators `Text Editor/Formatting/` `BreakBeforeTernaryOperators`Boolean	If enabled, ternary operators will be placed after line breaks.

Break Constructor Initializers Before Comma
`Text Editor/Formatting/`
`BreakConstructorInitializersBeforeCommaBool`
	If enabled, always break constructor initializers before commas and align the commas with the colon.

C++11 Braced List Style `Text Editor/Formatting/` `Cpp11BracedListStyle`Boolean	If enabled, format braced lists as best suited for C++11 braced lists.
Column Limit `Text Editor/Formatting/ColumnLimit`Integer	The column limit which limits the width of formatted lines.
Comment Pragmas `Text Editor/Formatting/CommentPragmas`String	A regular expression that describes comments with special meaning, which should not be split into lines or otherwise changed.

Constructor Initializer All On One Line Or One Per Line
`Text Editor/Formatting/`
`ConstructorInitializerAllOnOneLineOrOnePer`
	If enabled and the constructor initializers don't fit on a line, put each initializer on its own line.

Constructor Initializer Indent Width `Text Editor/Formatting/` `ConstructorInitializerIndentWidth`Integer	The number of characters to use for indentation of constructor initializer lists.
Continuation Indent Width `Text Editor/Formatting/` `ContinuationIndentWidth`Integer	Indent width for line continuations.
For-Each Macros `Text Editor/Formatting/` `ForEachMacros`StringList	A list of macros that should be interpreted as foreach loops rather than function calls.
Formatting Style `Text Editor/FormattingStyle`Enumeration	Select a set formatting options based on a named standard.
Indent Case Labels `Text Editor/Formatting/` `IndentCaseLabels`Boolean	If enabled, indent case labels one level from the switch statement.
Indent Width `Text Editor/Formatting/IndentWidth`Integer	The number of columns to use for indentation.

Indent Wrappend Function Names `Text Editor/Formatting/` `IndentWrappedFunctionNames`Boolean	If enabled, Indent if a function definition or declaration is wrapped after the type.
Keep Empty Lines At The Start Of Blocks `Text Editor/Formatting/` `KeepEmptyLinesAtTheStartOfBlocks`Boolean	If enabled, empty lines at the start of blocks are kept.
Maximum Empty Lines To Keep `Text Editor/Formatting/` `MaxEmptyLinesToKeep`Integer	The maximum number of consecutive empty lines to keep.
Namespace Indentation `Text Editor/Formatting/` `NamespaceIndentation`Enumeration	The indentation used for namespaces.
Penalty Break Before First Call Parameter `Text Editor/Formatting/` `PenaltyBreakBeforeFirstCallParameter`IntegerR	The penalty for breaking a function call after 'call('.
Penalty Break Before First Less-Less `Text Editor/Formatting/` `PenaltyBreakFirstLessLess`IntegerRange	The penalty for breaking before the first less-less.
Penalty Break Comment `Text Editor/Formatting/` `PenaltyBreakComment`IntegerRange	The penalty for each line break introduced inside a comment.
Penalty Break String `Text Editor/Formatting/` `PenaltyBreakString`IntegerRange	The penalty for each line break introduced inside a string literal.
Penalty Excess Character `Text Editor/Formatting/` `PenaltyExcessCharacter`IntegerRange	The penalty for each character outside of the column limit.
Penalty Return Type On Its Own Line `Text Editor/Formatting/` `PenaltyReturnTypeOnItsOwnLine`IntegerRange	Penalty for putting the return type of a function onto its own line.
Pointer Alignment `Text Editor/Formatting/` `PointerAlignment`Enumeration	Pointer and reference alignment style.
Space After C Style Cast `Text Editor/Formatting/` `SpaceAfterCStyleCast`Boolean	If enabled, a space may be inserted after C style casts.
Space Before Assignment Operators `Text Editor/Formatting/` `SpaceBeforeAssignmentOperators`Boolean	If disabled spaces will be removed before assignment operators.
Space Before Parentheses `Text Editor/Formatting/` `SpaceBeforeParens`Enumeration	Defines in which cases to put a space before opening parentheses.

Space In Empty Parentheses `Text Editor/Formatting/SpaceInEmptyParentheses`Boolean	If enabled, spaces may be inserted into '()'.
Spaces Before Trailing Comments `Text Editor/Formatting/SpacesBeforeTrailingComments`IntegerRange	The number of spaces before trailing line comments.
Spaces In Angles `Text Editor/Formatting/SpacesInAngles`Boolean	If enabled, spaces will be inserted around the angle brackets in template argument lists.
Spaces In C-style Cast Parentheses `Text Editor/Formatting/SpacesInCStyleCastParentheses`Boolean	If enabled, spaces may be inserted into C style casts.
Spaces In Container Literals `Text Editor/Formatting/SpacesInContainerLiterals`Boolean	If enabled, spaces are inserted inside container literals.
Spaces In Parentheses `Text Editor/Formatting/SpacesInParentheses`Boolean	If true, spaces will be inserted after '(' and before ')'.
Spaces In Square Brackets `Text Editor/Formatting/SpacesInSquareBrackets`Boolean	If true, spaces will be inserted after '[' and before ']'.
Standard `Text Editor/Formatting/Standard`Enumeration	Format compatible with this standard
Tab Style `Text Editor/Formatting/UseTab`Enumeration	The way to use hard tab characters in the resulting file.
Tab Width `Text Editor/Formatting/TabWidth`IntegerRange	The number of columns used for tab stops.

International

Property	**Description**
Default Text File Encoding `Text Editor/Default Codec`Enumeration	The encoding to use if not overridden by a project property or file is not in a known format.

Mouse

Property	**Description**
Alt+Left Click Action `Environment/Project Explorer/Alt+Left Click Action`Enumeration	The action the editor performs on Alt+Left Click

Alt+Middle Click Action `Environment/Project Explorer/Alt+Middle Click Action`Enumeration	The action the editor performs on Alt+Middle Click
Alt+Right Click Action `Environment/Project Explorer/Alt+Right Click Action`Enumeration	The action the editor performs on Alt+Right Click
Copy On Mouse Select `Text Editor/Copy On Mouse Select`Boolean	Automatically copy text to clipboard when marking a selection with the mouse.
Ctrl+Left Click Action `Environment/Project Explorer/Ctrl+Left Click Action`Enumeration	The action the editor performs on Ctrl+Left Click
Ctrl+Middle Click Action `Environment/Project Explorer/Ctrl+Middle Click Action`Enumeration	The action the editor performs on Ctrl+Middle Click
Ctrl+Right Click Action `Environment/Project Explorer/Ctrl+Right Click Action`Enumeration	The action the editor performs on Ctrl+Right Click
Middle Click Action `Environment/Project Explorer/Middle Click Action`Enumeration	The action the editor performs on Middle Click
Mouse Wheel Adjusts Font Size `Text Editor/Mouse Wheel Adjusts Font Size`Boolean	Enable or disable resizing of font by mouse wheel when CTRL key pressed.
Shift+Middle Click Action `Environment/Project Explorer/Shift+Middle Click Action`Enumeration	The action the editor performs on Shift+Middle Click
Shift+Right Click Action `Environment/Project Explorer/Shift+Right Click Action`Enumeration	The action the editor performs on Shift+Right Click

Programmer Assistance

Property	**Description**
ATTENTION Tag List `Text Editor/ATTENTION Tags`StringList	Set the tags to display as ATTENTION comments.
Ask For Index `Text Editor/Ask For Index`Boolean	Ask to index the project if goto symbol fails in current editor context.
Auto-Comment Text `Text Editor/Auto Comment`Boolean	Enable or disable automatically swapping commenting on source lines by typing '/' with an active selection.

Auto-Surround Text `Text Editor/Auto Surround`Boolean	Enable or disable automatically surrounding selected text when typing triangular brackets, quotation marks, parentheses, brackets, or braces.
Check Spelling `Text Editor/Spell Checking`Boolean	Enable spell checking in comments.
Display Code Completion Suggestions While Typing `Text Editor/Suggest Completion While Typing`Boolean	Enable code completion as you type without needing to use the show suggestions key (Ctrl+J).
Enable Popup Diagnostics `Text Editor/Enable Popup Diagnostics`Boolean	Enables on-screen diagnostics in the text editor.
FIXME Tag List `Text Editor/FIXME Tags`StringList	Set the tags to display as FIXME comments.
Include Preprocessor Definitions in Suggestions `Text Editor/Preprocessor Definition Suggestions`Boolean	Include or exclude preprocessor definitions in code completion suggestions.
Include Templates in Suggestions `Text Editor/Template Suggestions`Boolean	Include or exclude templates in code completion suggestions.
Lint Tag List `Text Editor/LINT Tags`StringList	Set the tags to display as Lint directives.
Template Characters To Match `Text Editor/Template Suggestions Characters`IntegerRange	The number of characters to match before suggesting a template.

Save

Property	**Description**
Backup File History Depth `Text Editor/Backup File Depth`IntegerRange	The number of backup files to keep when saving an existing file.
Delete Trailing Space On Save `Text Editor/Delete Trailing Space On Save`Boolean	Deletes trailing whitespace from each line when a file is saved.
Tab Cleanup On Save `Text Editor/Cleanup Tabs On Save`Enumeration	Cleans up tabs when a file is saved.

Visual Appearance

Property	**Description**
Font `Text Editor/Font`FixedPitchFont	The font to use for text editors.

Font Smoothing Threshold `Text Editor/Antialias Threshold`IntegerRange	The minimum size for font smoothing: font sizes smaller than this will have antialiasing turned off.
Hide Cursor When Typing `Text Editor/Hide Cursor When Typing`Boolean	Hide or show the I-beam cursor when you start to type.
Highlight Cursor Line `Text Editor/Highlight Cursor Line`Boolean	Enable or disable visually highlighting the cursor line.
Horizontal Scroll Bar `Text Editor/HScroll Bar`Enumeration	Show or hide the horizontal scroll bar.
Insert Caret Style `Text Editor/Insert Caret Style`Enumeration	How the caret is displayed with the editor in insert mode.
Line Numbers `Text Editor/Line Number Mode`Enumeration	How often line numbers are displayed in the margin.
Mate Matching Mode `Text Editor/Mate Matching Mode`Enumeration	Controls when braces, brackets, and parentheses are matched.
Overwrite Caret Style `Text Editor/Overwrite Caret Style`Enumeration	How the caret is displayed with the editor in overwrite mode.
Show Diagnostic Icons In Gutter `Text Editor/Diagnostic Icons`Boolean	Enables display of diagnostic icons in the icon gutter.
Show Icon Gutter `Text Editor/Icon Gutter`Boolean	Show or hide the left-hand gutter containing breakpoint, bookmark, and optional diagnostic icons.
Show Mini Toolbar `Text Editor/Mini Toolbar`Boolean	Show the mini toolbar when selecting text with the mouse.
Use I-beam Cursor `Text Editor/Ibeam cursor`Boolean	Show an I-beam or arrow cursor in the text editor.
Vertical Scroll Bar `Text Editor/VScroll Bar`Enumeration	Show or hide the vertical scroll bar.

Windows Environment Options

Call Stack

Property	Description
Execution Frame at Top `Environment/Call Stack/Most Recent At Top`Boolean	Controls whether the most recent call is at the top or the bottom of the list.
Show Call Address `Environment/Call Stack/Show Call Address`Boolean	Enables the display of the call address in the call stack.
Show Call Source Location `Environment/Call Stack/Show Call Location`Boolean	Enables the display of the call source location in the call stack.
Show Frame Size `Environment/Call Stack/Show Stack Usage`Boolean	Enables the display of the amount of stack used by the call.
Show Frame Size In Bytes `Environment/Call Stack/Show Stack Usage In Bytes`Boolean	Display the stack usage in bytes rather than words.
Show Parameter Names `Environment/Call Stack/Show Parameter Names`Boolean	Enables the display of parameter names in the call stack.
Show Parameter Types `Environment/Call Stack/Show Parameter Types`Boolean	Enables the display of parameter types in the call stack.
Show Parameter Values `Environment/Call Stack/Show Parameter Values`Boolean	Enables the display of parameter values in the call stack.
Show Stack Pointer `Environment/Call Stack/Show Stack Pointer`Boolean	Enables the display of the stack pointer in the call stack.
Show Stack Usage `Environment/Call Stack/Show Cumulative Stack Usage`Boolean	Enables the display of the amount of stack used.
Show Stack Usage In Bytes `Environment/Call Stack/Show Cumulative Stack Usage In Bytes`Boolean	Display the stack usage in bytes rather than words.

Clipboard Ring

Property	Description
Maximum Items Held In Ring `Environment/Clipboard Ring/Max Entries`IntegerRange	The maximum number of items held on the clipboard ring before they are recycled.
Preserve Contents Between Runs `Environment/Clipboard Ring/Save`Boolean	Save the clipboard ring across SEGGER Embedded Studio runs.

Outline Window

Property	Description
Group #define Directives `Windows/Outline/Group Defines`Boolean	Group consecutive #define and #undef preprocessor directives.
Group #if Directives `Windows/Outline/Group Ifs`Boolean	Group lines contained betwen #if, #else, and #endif preprocessor directives.
Group #include Directives `Windows/Outline/Group Includes`Boolean	Group consecutive #include preprocessor directives.
Group Top-Level Declarations `Windows/Outline/Group Top Level Items`Boolean	Group consecutive top-level variable and type declarations.
Group Visibility `Windows/Outline/Group Visibility`Boolean	Group class members by public, protected, and private visibility.
Hide #region Prefix `Windows/Outline/Hide Region Prefix`Boolean	Hides the '#region' prefix from groups and shows only the group name.
Refresh Outline and Preview `Windows/Outline/Preview Refresh Mode`Enumeration	How the Preview pane refreshes its contects.

Project Explorer

Property	Description
Add Filename Replace Macros `Environment/Project Explorer/Filename Replace Macros`StringList	Macros (system and global) used to replace the start of a filename on project file addition.
Color Project Nodes `Environment/Project Explorer/Color Nodes`Boolean	Show the project nodes colored for identification in the Project Explorer.
Confirm Configuration Folder Delete `Project Explorer/Confirm Configuration Folder Delete`Boolean	Display a confirmation prompt before deleting a configuration folder cotaining properties.

Confirm Forget Modified Options `Project Explorer/Confirm Reject Property Changes`Boolean	Display a confirmation prompt before forgetting option modifications.
Context Menu Edit Options At Top `Environment/Project Explorer/Context Menu Properties Position`Boolean	Controls where Edit Options are displayed by the Project Explorer's context menu.
Context Menu Uses Common Folder `Environment/Project Explorer/Context Menu Common Folder`Boolean	Controls how common options are displayed by the Project Explorer's context menu.
External Editor `Environment/Project Explorer/External Editor`FileName	The file name of the application to use as the external text editor
Favorite Properties `Environment/Project Explorer/Favorite Properties`StringList	The favorite list of properties that are displayed starred and before other properties in the Project Explorer.
Highlight Dynamic Items `Environment/Project Explorer/Show Dynamic Overlay`Boolean	Show an overlay on an item if it is populated from a dynamic folder.
Highlight External Items `Environment/Project Explorer/Show Non-Local Overlay`Boolean	Show an overlay on an item if it is not held within the project directory.
Output Files Folder `Environment/Project Explorer/Show Output Files`Boolean	Show the build output files in an Output Files folder in the project explorer.
Read-Only Data In Code `Environment/Project Explorer/Statistics Read-Only Data Handling`Boolean	Configures whether read-only data contributes to the Code or Data statistic.
Show Dependencies `Environment/Project Explorer/Dependencies Display`Enumeration	Controls how the dependencies are displayed.
Show Favorite Properties `Environment/Project Explorer/Context Menu Show Favorites`Boolean	Controls if favorite properties are displayed by the Project Explorer's context menu.
Show File Count on Folder `Environment/Project Explorer/Count Files`Boolean	Show the number of files contained in a folder as a badge in the Project Explorer.
Show Modified Options on Folder/File `Environment/Project Explorer/Show Modified Properties`Boolean	Show if a folder or file has modified options as a badge in the Project Explorer.
Show Options `Environment/Project Explorer/Properties Display`Enumeration	Controls how the options are displayed.

Show Project Count on Solution `Environment/Project Explorer/Count Projects`Boolean	Show the number of projects contained in a solution as a badge in the Project Explorer.
Show Source Control Annotation `Environment/Project Explorer/Show Source Control Annotation`Boolean	Annotate items in the project explorer with their source control status.
Show Statistics Rounded `Environment/Project Explorer/Statistics Format`Boolean	Show exact or rounded sizes in the project explorer.
Source Control Status Column `Environment/Project Explorer/Show Source Control Column`Boolean	Show the source control status column in the project explorer.
Starred Files Names `Environment/Project Explorer/Starred File Names`StringList	The list of wildcard-matched file names that are highligted with stars, to bring attention to themselves, in the Project Explorer.
Statistics Column `Environment/Project Explorer/Statistics Display`Boolean	Show the code and data size columns in the Project Explorer.
Synchronize Explorer With Editor `Environment/Project Explorer/Sync Editor`Boolean	Synchronizes the Project Explorer with the document being edited.
Use Common Options Folder `Environment/Project Explorer/Common Properties Display`Boolean	Controls how common options are displayed.

Variable Window

Property	Description
Show Variable Address Column `Environment/Variable Window/Show Address Column`Boolean	Controls whether the variable address column is displayed.
Show Variable Size Column `Environment/Variable Window/Show Size Column`Boolean	Controls whether the variable size column is displayed.
Show Variable Type Column `Environment/Variable Window/Show Type Column`Boolean	Controls whether the variable type column is displayed.

Windows Window

Property	Description

Buffer Grouping `Environment/Windows/Grouping`Enumeration	How the files are grouped or listed in the Windows window.
Show File Path as Tooltip `Environment/Windows/Show Filename Tooltips`Boolean	Show the full file name as a tooltip when hovering over files in the Windows window.
Show Line Count and File Size `Environment/Windows/Show Sizes`Boolean	Show the number of lines and size of each file in the windows list.

General Build Options

Build

Property	Description
Always Rebuild `build_always_rebuild`Boolean	Specifies whether or not to always rebuild the project/folder/file.
Batch Build Configurations `batch_build_configurations`StringList	The set of configurations to batch build.
Build Quietly `build_quietly`Boolean	Suppress the display of startup banners and information messages.
Enable Unused Symbol Removal `build_remove_unused_symbols`Boolean	Enable the removal of unused symbols from the executable.
Exclude From Build `build_exclude_from_build`Boolean	Specifies whether or not to exclude the project/folder/file from the build.
Include Debug Information `build_debug_information`Boolean	Specifies whether symbolic debug information is generated.
Intermediate Directory `build_intermediate_directory`DirPath	Specifies a relative path to the intermediate file directory. This property will have macro expansion applied to it. The macro $(IntDir) is set to this value.
Memory Map File `linker_memory_map_file`ProjFileName	The name of the file containing the memory map description.
Memory Map Macros `linker_memory_map_macros`StringList	Macro values to substitue in memory map nodes. Each macro is defined as **name=value** and are seperated by ;.
Output Directory `build_output_directory`DirPath	Specifies a relative path to the output file directory. This property will have macro expansion applied to it. The macro $(OutDir) is set to this value. The macro $(RootRelativeOutDir) is set relative to the **Root Output Directory** if specified.
Project Can Build In Parallel `project_can_build_in_parallel`Enumeration	Specifies that dependent projects can be built in parallel. Default is No for Staging and Combining project types, Yes for all other project types.
Project Dependencies `project_dependencies`StringList	Specifies the projects the current project depends upon.
Project Directory `project_directory`String	Path of the project directory relative to the directory containing the project file. The macro $(ProjectDir) is set to the absolute path of this property.

Project Macros `macros`StringList	Specifies macro values which are expanded in project properties and for file names in Common configuration only. Each macro is defined as **name=value** and are seperated by **;**.
Project Type `project_type`Enumeration	Specifies the type of project to build. The options are **Executable, Library, Object file, Staging, Combining, Externally Built Executable, Externally Built Library**.
Property Groups File `property_groups_file_path`ProjFileName	The file containing the property groups for this project. This is applicable to **Executable** and **Externally Built Executable** project types only.
Root Output Directory `build_root_output_directory`DirPath	Allows a common root output directory to be specified that can be referenced using the $(RootOutDir) macro.
Suppress Warnings `build_suppress_warnings`Boolean	Don't report warnings.
Tool Chain Directory `build_toolchain_directory`DirPath	Specify the root of the toolchain directory. This property will have macro expansion applied to it. The macro $(ToolChainDir) is set to this value.
Treat Warnings as Errors `build_treat_warnings_as_errors`Boolean	Treat all warnings as errors.

Combining

Property	Description
Combine Command `combine_command`Unknown	The command to execute. This property will have macro expansion applied to it with the macro **$(CombiningOutputFilePath)** set to the output filepath of the combine command and the macro **$(CombiningRelInputPaths)** is set to the (project relative) names of all of the files in the project.
Combine Command Working Directory `combine_command_wd`String	The working directory in which the combine command is run. This property will have macro expansion applied to it.
Output File Path `combine_output_filepath`String	The output file path the stage command will create. This property will have macro expansion applied to it.
Set To Read-only `combine_set_readonly`Enumeration	Set the output file to read only or read/write.

External Build

Property	Description

Archive Command `external_archive_command`Unknown	The command line to archive object files. This property will have macro expansion applied to it with the additional macros: $(TargetPath) contains the full file name of the **Library File Name** property $(RelTargePath) contains the project directory relative file name of the **Object File Name** property. $(Objects) a space seperated list of files to archive, generated from the source files of the project OR. $(ObjectsFilePath) contains the full file name of the file containing the list of files to link
Assemble Command `external_assemble_command`Unknown	The command line to assemble an assembly source file. This property will have macro expansion applied to it with the additional macros: $(TargetPath) contains the full file name of the **Object File Name** property. $(RelTargePath) contains the project directory relative file name of the **Object File Name** property. $(AsmOptions) contains a space seperated list of options as set in the **Additional Assembler Options** property. $(DependencyPath) contains the filename of the .d file that is required to be output by the compilation for dependency support. $(Defines) contains a space seperated list of preprocessor definitions as set in the **Preprocessor Definitions** propety. $(Includes) contains a space seperated list of user include directories as set in the **User Include Directories** property.
Build Command `external_build_command`Unknown	The command line to build the executable e.g. make. This property will have macro expansion applied to it.

C Compile Command `external_c_compile_command`Unknown	The command line to compile a C source file. This property will have macro expansion applied to it with the additional macros: $(TargetPath) contains the full file name of the **Object File Name** property. $(RelTargePath) contains the project directory relative file name of the **Object File Name** property. $(COptions) contains a space seperated list of options as set in the **C Additional C/C++ Compiler Options** property. $(COnlyOptions) contains a space seperated list of options as set in the **C Additional C Compiler Only Options** property. $(DependencyPath) contains the filename of the .d file that is required to be output by the compilation for dependency support. $(Defines) contains a space seperated list of preprocessor definitions as set in the **Preprocessor Definitions** propety. $(Includes) contains a space seperated list of user include directories as set in the **User Include Directories** property.
C++ Compile Command `external_cpp_compile_command`Unknown	The command line to compile a C++ source file. This property will have macro expansion applied to it with the additional macros: $(TargetPath) contains the full file name of the **Object File Name** property. $(RelTargePath) contains the project directory relative file name of the **Object File Name** property. $(COptions) contains a space seperated list of options as set in the **C Additional C/C++ Compiler Options** property. $(CppOnlyOptions) contains a space seperated list of options as set in the **C Additional C++ Compiler Only Options** property. $(DependencyPath) contains the filename of the .d file that is required to be output by the compilation for dependency support. $(Defines) contains a space seperated list of preprocessor definitions as set in the **Preprocessor Definitions** propety $(Includes) contains a space seperated list of user include directories as set in the **User Include Directories** property.

Clean Command `external_clean_command`Unknown	The command line to clean the executable e.g. make clean. This property will have macro expansion applied to it.
Link Command `external_link_command`Unknown	The command line to link an executable. This property will have macro expansion applied to it with the additional macros: $(TargetPath) contains the full file name of the **Executable File Name** property. $(RelTargePath) contains the project directory relative file name of the **Executable File Name** property. $(LinkOptions) contains a space seperated list of options as set in the **Additional Linker Options** property. $(Objects) a space seperated list of files to link, generated from the source files of the project and the outputs of any dependent projects OR. $(ObjectsFilePath) contains the full file name of the file containing the list of files to link
Objects File `external_objects_file_name`Unknown	The name of the file containing the list of files to archive or link, generated from the source files of the project.This property will have macro expansion applied to it. The macro $(ObjectsFilePath) is set to this value.

File

Property	**Description**
File Encoding `file_codec`Enumeration	Specifies the encoding to use when reading and writing the file.
File Name `file_name`String	The name of the file. This property will have global macro expansion applied to it. The following macros are set based on the value: $(InputDir) relative directory of file, $(InputName) file name without directory or extension, $(InputFileName) file name, $(InputExt) file name extension, $(InputPath) absolute path to the file name, $(RelInputPath) relative path from project directory to the file name.
File Open Action `file_open_with`Enumeration	Specifies how to open the file when it is double clicked.
File Type `file_type`Enumeration	The type of file. Default setting uses the file extension to determine file type.

Flag `file_flag`Enumeration	Flag which you can use to draw attention to important files in your project.

Folder

Property	Description
Dynamic Folder Directory `path`DirPath	Dynamic folder directory specification.
Dynamic Folder Exclude `exclude`StringList	Dynamic folder exclude specification - ; seperated wildcards.
Dynamic Folder Filter `filter`String	Dynamic folder filter specification - ; seperated wildcards.
Dynamic Folder Recurse `recurse`Boolean	Dynamic folder recurse into subdirectories.
Unity Build Exclude Filter `unity_build_exclude_filter`String	The filter specification to exclude from the unity build - ; seperated wildcards.
Unity Build File Name `unity_build_file_name`FileName	The file name created that #includes all files in the folder for the unity build.

General

Property	Description
Inherited Configurations `inherited_configurations`StringList	The list of configurations that are inherited by this configuration.

Library

Property	Description
Library File Name `build_output_file_name`FileName	Specifies a name to override the default library file name.
Use Indirect File `arm_archiver_indirect_file`Boolean	Create indirect file for input files.

Package

Property	Description
Package Dependencies `package_dependencies`StringList	Specifies the packages the current project depends upon.

Project

Property	Description
Flag `project_flag`Enumeration	Flag which you can use to draw attention to important projects in your solution.

Solution

Property	Description
Flag `solution_flag`Enumeration	Flag which you can use to draw attention to important projects in your solution.

Source Code

Property	Description
Inhibit Source Indexing `project_inhibit_indexing`Boolean	Disable source indexing for projects that would normally be indexed (executable and library projects).

Staging

Property	Description
Output File Path `stage_output_filepath`String	The output file path the stage command will create. This property will have macro expansion applied to it.
Set To Read-only `stage_set_readonly`Enumeration	Set the output file permissions to read only or read/write.
Stage Command `stage_command`Unknown	The command to execute. This property will have macro expansion applied to it with the additional **$(StageOutputFilePath)** macro set to the output filepath of the stage command.
Stage Command Working Directory `stage_command_wd`String	The working directory in which the stage command is run. This property will have macro expansion applied to it.
Stage Post-Build Command `stage_post_build_command`Unknown	The command to execute after staging commands have executed. This property will have macro expansion applied to it.
Stage Post-Build Command Working Directory `stage_post_build_command_wd`String	The working directory where the post build command runs. This property will have macro expansion applied to it.

Compilation Options

Assembler

Property	Description
Additional Assembler Options `asm_additional_options`StringList	Enables additional options to be supplied to the assembler. This property will have macro expansion applied to it.
Additional Assembler Options From File `asm_additional_options_from_file`ProjFileName	Enables additional options to be supplied to the assembler from a file. This property will have macro expansion applied to it.
Assembler `arm_assembler_variant`Enumeration	Specifies which assembler to use.

Code Generation

Property	Description
ARM Advanced SIMD Type `arm_advanced_SIMD_type`Enumeration	Specifies the Advanced SIMD type to generate code for. The options are: NEON - Cortex-A based processors

ARM Architecture `arm_architecture`Enumeration	Specifies the version of the instruction set to generate code for. The options are: v4T - ARM7TDMI and ARM920T processors v5TE - ARM9E, Feroceon and XScale processors v6 - ARM11 processors v6M - Cortex-M0 and Cortex-M1 processors v7M - Cortex-M3 processors v7EM - Cortex-M4 and Cortex-M7 processors v7R - Cortex-R processors v7A - Cortex-A processors v8M_Baseline v8M_Mainline The corresponding preprocessor definitions: __ARM_ARCH_4T__ __ARM_ARCH_5TE__ __ARM_ARCH_6__ __ARM_ARCH_6M__ __ARM_ARCH_7M__ __ARM_ARCH_7EM__ __ARM_ARCH_7R__ __ARM_ARCH_7A__ __ARM_ARCH_8M_BASELINE__ __ARM_ARCH_8M_MAINLINE__ are defined.
ARM Core Type `arm_core_type`Enumeration	Specifies the core to generate code for. The options are: ARM7TDMI, ARM7TDMI-S, ARM720T ARM920T, ARM946E-S, ARM966E-S, ARM968E-S, ARM926EJ-S ARM1136J-S, ARM1136JF-S, ARM1176JZ-S, ARM1176JZF-S Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, Cortex-M4, Cortex-M7, v8M_Baseline, v8M_Mainline Cortex-R4, Cortex-R4F, Cortex-R5 Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9 XScale None If this property is set to None then the architecture property is used

ARM FP ABI Type `arm_fp_abi`Enumeration	Specifies the FP ABI type to generate code for. The options are: **Soft** generate calls to the C library to implement floating point operations. **SoftFP** generate VFP code to implement floating point operations. **Hard** generate VFP code to implement floating point operations and use VFP registers to pass floating point parameters on function calls. **None** will not specify the FP ABI or the FPU.
ARM FPU Type `arm_fpu_type`Enumeration	Specifies the FPU type to generate code for. The options are: VFP - ARM9/ARM11 based processors VFP9 - the same as VFP VFPv3-D32 - Cortex-A/Cortex-R based processors VFPv3-D16 - Cortex-A/Cortex-R based processors VFPv4-D32 - Cortex-A/Cortex-R based processors VFPv4-D16 - Cortex-A/Cortex-R based processors FPv4-SP-D16 - Cortex-M4 processors FPv5-SP-D16 - Cortex-M7 processors FPv5-D16 - Cortex-M7 processors The corresponding preprocessor definitions: __ARM_ARCH_VFP__ __ARM_ARCH_VFP3_D32__ __ARM_ARCH_VFP3_D16__ __ARM_ARCH_VFP4_D32__ __ARM_ARCH_VFP4_D16__ __ARM_ARCH_FPV4_SP_D16__ __ARM_ARCH_FPV5_SP_D16__ __ARM_ARCH_FPV5_D16__ are defined.
ARM/Thumb Interworking `arm_interwork`Enumeration	Specifies whether ARM/Thumb interworking code should be generated. Setting this property to **No** may result in smaller code sizes when compiling for architecture **v4T**.
Byte Order `arm_endian`Enumeration	Specify the byte order of the target processor.
CM0/CM0+/CM1 Has Small Multiplier `arm_cm0_has_small_multiplier`Boolean	The CM0/CM0+/CM1 core has the small multiplier.
Debugging Level `gcc_debugging_level`Enumeration	Specifies the level of debugging information to generate.
Dwarf Version `gcc_dwarf_version`Enumeration	Specifies the version of Dwarf debugging information to generate.

Emit Assembler CFI `gcc_emit_assembler_cfi`Boolean	Emit DWARF 2 unwind info using GAS .cfi_* directives rather than a compiler generated .eh_frame section.
Enable All Warnings `gcc_enable_all_warnings`Boolean	Enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
Enable Exception Support `cpp_enable_exceptions`Boolean	Specifies whether exception support is enabled for C++ programs.
Enable RTTI Support `cpp_enable_rtti`Boolean	Specifies whether RTTI support is enabled for C++ programs.
Enumeration Size `gcc_short_enum`Enumeration	Select between minimal container sized enumerations and int sized enumerations.
Instruction Set `arm_instruction_set`Enumeration	Specifies the instruction set to generate code for.
Instrument Functions `arm_instrument_functions`Boolean	Specifies whether instrumentation calls are generated for function entry and exit.
Long Calls `arm_long_calls`Boolean	Specifies whether function calls are made using absolute addresses.
Merge Globals [clang] `clang_merge_globals`Boolean	Select whether global declarations are merged. This may reduce code size and increase execution speed for some applications. However, if functions are not used in an application and are eliminated by the linker, merged globals may increase the data size requirement of an application.
No COMMON `gcc_no_common`Boolean	Don't put globals in the common section
Omit Frame Pointer `gcc_omit_frame_pointer`Boolean	Specifies whether a frame pointer register is omitted if not required.
Optimization Level `gcc_optimization_level`Enumeration	Specifies the optimization level to use.
Treat 'double' as 'float' `double_is_float`Boolean	Forces the compiler to make 'double' equivalent to 'float'.
Use Builtins `arm_use_builtins`Boolean	Use built-in library functions e.g. scanf
V7A/V7R Has Integer Divide Instructions `arm_v7_has_divide_instructions`Boolean	The V7A/V7R architecture has integer divide instructions in both ARM and Thumb instruction sets.
Wide Character Size `gcc_wchar_size`Enumeration	Select between standard 32-bit or shorter 16-bit size for wide characters and wchar_t.

Compiler

Property	Description
Additional C Compiler Only Options `c_only_additional_options`StringList	Enables additional options to be supplied to the C compiler only. This property will have macro expansion applied to it.
Additional C Compiler Only Options From File `c_only_additional_options_from_file`ProjFileNa	Enables additional options to be supplied to the C compiler only from a file. This property will have macro expansion applied to it.
Additional C++ Compiler Only Options `cpp_only_additional_options`StringList	Enables additional options to be supplied to the C++ compiler only. This property will have macro expansion applied to it.
Additional C++ Compiler Only Options From File `cpp_only_additional_options_from_file`ProjFile	Enables additional options to be supplied to the C++ compiler only from a file. This property will have macro expansion applied to it.
Additional C/C++ Compiler Options `c_additional_options`StringList	Enables additional options to be supplied to the C/C++ compiler. This property will have macro expansion applied to it.
Additional C/C++ Compiler Options From File `c_additional_options_from_file`ProjFileName	Enables additional options to be supplied to the C/C++ compiler from a file. This property will have macro expansion applied to it.
C Language Standard `gcc_c_language_standard`Enumeration	Specifies the language standard to use when compiling C files.
C++ Language Standard `gcc_cplusplus_language_standard`Enumeration	Specifies the language standard to use when compiling C files.
Compiler `arm_compiler_variant`Enumeration	Specifies which compiler to use.
Enforce ANSI Checking `c_enforce_ansi_checking`Boolean	Perform additional checks for ensure strict conformance to the selected ISO (ANSI) C or C++ standard.
Keep Assembly Source `arm_keep_assembly`Boolean	Specifies whether assembly code generated by the compiler is kept.
Keep Preprocessor Output `arm_keep_preprocessor_output`Boolean	Specifies whether preprocessor output generated by the compiler is kept.
Object File Name `build_object_file_name`FileName	Specifies a name to override the default object file name.
Supply Absolute File Path `arm_supply_absolute_file_path`Boolean	Specifies whether absolute file paths are supplied to the compiler.

Preprocessor

Property	Description
Ignore Includes `c_ignore_includes`Boolean	Ignore the include directories properties.
Preprocessor Definitions `c_preprocessor_definitions`StringList	Specifies one or more preprocessor definitions. This property will have macro expansion applied to it.
Preprocessor Undefinitions `c_preprocessor_undefinitions`StringList	Specifies one or more preprocessor undefinitions. This property will have macro expansion applied to it.
System Include Directories `c_system_include_directories`StringList	Specifies the system include path. This property will have macro expansion applied to it.
Undefine All Preprocessor Definitions `c_undefine_all_preprocessor_definitions`Bool	Does not define any standard preprocessor definitions.
User Include Directories `c_user_include_directories`StringList	Specifies the user include path. This property will have macro expansion applied to it.

Section

Property	Description
Code Section Name `default_code_section`String	Specifies the default name to use for the program code section.
Constant Section Name `default_const_section`String	Specifies the default name to use for the read-only constant section.
Data Section Name `default_data_section`String	Specifies the default name to use for the initialized, writable data section.
ISR Section Name `default_isr_section`String	Specifies the default name to use for the ISR code.
Vector Section Name `default_vector_section`String	Specifies the default name to use for the interrupt vector section.
Zeroed Section Name `default_zeroed_section`String	Specifies the default name to use for the zero-initialized, writable data section.

User Build Step

Property	Description
Post-Compile Command `compile_post_build_command`Unknown	A command to run after the compile command has completed. This property will have macro expansion applied to it with the additional **$(TargetPath)** macro set to the output filepath of the compiler command.

Post-Compile Working Directory `compile_post_build_command_wd`DirPath	The working directory where the post-compile command is run. This property will have macro expansion applied to it.
Pre-Compile Command `compile_pre_build_command`Unknown	A command to run before the compile command. This property will have macro expansion applied to it.
Pre-Compile Command Output File Path `compile_pre_build_command_output_file_name`	The pre-compile generated file name. This property will have macro expansion applied to it.
Pre-Compile Working Directory `compile_pre_build_command_wd`DirPath	The working directory where the pre-compile command is run. This property will have macro expansion applied to it.

Debugging Options

Debugger

Property	Description
Command Arguments `debug_command_arguments`String	The command arguments passed to the executable. This property will have macro expansion applied to it.
DABORT Handler Name `dabortHandler_name`String	The name of the dabort handler symbol. Used for backtracing out of exception handlers.
Debug Dependent Projects `debug_dependent_projects`Boolean	Debugger will debug dependent projects.
Debug Symbols File `external_debug_symbols_file_name`ProjFileName	The name of the debug symbols file. This property will have macro expansion applied to it. If it is not defined then the main load file is used.
Debug Symbols Load Address `external_debug_symbols_load_address`String	The (code) address to be added to the debug symbol (code) addresses.
Default debugIO implementation `arm_debugIO_Implementation`Enumeration	The default debugIO implementation.
Entry Point Symbol `debug_entry_point_symbol`String	Debugger will start execution at symbol if defined.
FIQ Handler Name `fiqHandler_name`String	The name of the fiq handler symbol. Used for backtracing out of exception handlers.
IRQ Handler Name `irqHandler_name`String	The name of the irq handler symbol. Used for backtracing out of exception handlers.
Ignore .debug_aranges Section `debug_ignore_debug_aranges`Boolean	The debugger will not use the .debug_aranges section.
Ignore .debug_frame Section `debug_ignore_debug_frame`Boolean	The debugger will not use the .debug_frame section.
Initial Breakpoint Is Set `debug_initial_breakpoint_set_option`Enumerati	Specify when the initial breakpoint should be set
Leave Target Running `debug_leave_target_running`Boolean	Debugger will leave the target running on debug stop.
Load Offset `debug_load_file_offset`String	The offset to add to the load address of the load file.This offset is added to any absolute relocations of symbols (whose address is less than Load Offset Symbol Limit) if the load file contains relocation sections.
Load Offset Symbol Limit `debug_load_file_limit`String	If set apply the Load Offset logic to only those symbols that have addresses less than the specified limit.

PABORT Handler Name `pabortHandler_name`String	The name of the pabort handler symbol. Used for backtracing out of exception handlers.
Register Definition File `debug_register_definition_file`ProjFileName	The name of the file containing register definitions.
Reserved Member Name `reservedMember_name`String	The struct reserved member name. Struct members that contain the (case insensitive) string will not be displayed.
Run To `debug_initial_breakpoint`String	An initial breakpoint to set if no other breakpoints exist
SWI Handler Name `swiHandler_name`String	The name of the swi handler symbol. Used for backtracing out of exception handlers.
Start Address `external_start_address`String	The address to start the externally built executable running from.
Start From Entry Point Symbol `debug_start_from_entry_point_symbol`Boolean	If yes the debugger will start execution from the entry point symbol.If no the debugger will start execution from the core specific location.
Startup Completion Point `debug_startup_completion_point`String	Specifies the point in the program where startup is complete. Software breakpoints and debugIO will be enabled after this point has been reached.
Target Connection `debug_target_connection`Enumeration	Specifies the target to connect to for debugging actions.
Thread Maximum `debug_threads_max`IntegerRange	The maximum number of threads to display.
Threads Script File `debug_threads_script`ProjFileName	The threads script used by the debugger.
Type Interpretation File `debug_type_file`FileName	Specifies the type interpretation file to use.
UNDEF handler name `undefHandler_name`String	The name of the undef handler symbol. Used for backtracing out of exception handlers.
Working Directory `debug_working_directory`DirPath	The working directory for a debug session. This property will have macro expansion applied to it.

J-Link

Property	**Description**
Additional J-Link Options `JLinkExecuteCommand`StringList	Specify additional J-Link options to allow enabling or disabling advanced features and fine tuning. For more information see **J-Link Command Strings**
Enable Adaptive Clocking `adaptive`Enumeration	Adaptive clocking is enabled.

Exclude Flash Cache Range `JLinkExcludeFlashCacheRangeString`	Define a memory range that should not be cached by J-Link. By default all areas that J-Link knows to be Flash memory are cached. This means that it is assumed that the contents of these areas do not change during program execution. If this assumption does not hold true, typically because the target program modifies the flash content for data storage, then the affected area should be excluded from the cache. This may slightly reduce the debugging speed. Syntax: either 'start_address-end_address' or 'address,size'. For example: 0x08000000,0x1000.
Host Connection `ConnectionEnumeration`	Defines how to connect the host to the J-Link: "USB": Connect to J-Link via USB "USB S/N": Connect to J-Link with specified serial number via USB e.g. USB 174300001 "IP S/N": Connect to J-Link with specified serial number via IP e.g. IP 174300001 "IP n.n.n.n": Connect to J-Link with specified IP address e.g. IP 192.168.20.20
JTAG Instruction Register Size Before Target `arm_linker_jtag_pad_post_irIntegerRange`	Specifies the number of bits in the instruction register before the target (as seen from TDI), which is the number of bits to pad the JTAG instruction register with the BYPASS instruction after the target instruction.
JTAG Number Of Devices Before Target `arm_linker_jtag_pad_post_drIntegerRange`	Specifies the number of devices before the target (as seen from TDI), which is the number of bits to pad the JTAG data register.
Log File `JLinkLogFileNameFileName`	The file to output the J-Link log to.
Script File `JLinkScriptFileNameFileName`	The file path of the optional J-Link script file to use.
Show Log Messages In Output Window `showLogEnumeration`	Display the J-Link log messages to the output window.
Speed `speedIntegerRange`	The required JTAG/SWD clock frequency in kHz (0 to auto-detect best possible).
Supply Power `supplyPowerEnumeration`	The J-Link supplies power to the target.
Target Interface Type `arm_target_interface_typeEnumeration`	Specifies the type of interface the target has. The options are: JTAG - Use JTAG interface SWD - Use SWD interface

Loader

Property	Description
Additional Load File Address[0] `debug_additional_load_file_address`String	The address to load the additional load file.
Additional Load File Address[1] `debug_additional_load_file_address1`String	The address to load the additional load file.
Additional Load File Address[2] `debug_additional_load_file_address2`String	The address to load the additional load file.
Additional Load File Address[3] `debug_additional_load_file_address3`String	The address to load the additional load file.
Additional Load File Type[0] `debug_additional_load_file_type`Enumeration	The file type of the additional load file. The options are **Detect, elf, bin, ihex, hex, tihex, srec**.
Additional Load File Type[1] `debug_additional_load_file_type1`Enumeration	The file type of the additional load file. The options are **Detect, elf, bin, ihex, hex, tihex, srec**.
Additional Load File Type[2] `debug_additional_load_file_type2`Enumeration	The file type of the additional load file. The options are **Detect, elf, bin, ihex, hex, tihex, srec**.
Additional Load File Type[3] `debug_additional_load_file_type3`Enumeration	The file type of the additional load file. The options are **Detect, elf, bin, ihex, hex, tihex, srec**.
Additional Load File[0] `debug_additional_load_file`ProjFileName	Additional file to load on debug load. This property will have macro expansion applied to it.
Additional Load File[1] `debug_additional_load_file1`ProjFileName	Additional file to load on debug load. This property will have macro expansion applied to it.
Additional Load File[2] `debug_additional_load_file2`ProjFileName	Additional file to load on debug load. This property will have macro expansion applied to it.
Additional Load File[3] `debug_additional_load_file3`ProjFileName	Additional file to load on debug load. This property will have macro expansion applied to it.
Load ELF Sections `debug_load_sections`Enumeration	The debugger will load ELF sections rather than ELF programs.
Load File `external_build_file_name`ProjFileName	The name of the main load file. This property will have macro expansion applied to it. If it is not defined then the output filepath of the linker command is used.
Load File Address `external_load_address`String	The address to download the main load file to.
Load File Type `external_load_file_type`Enumeration	The file type of the main load file. The options are **Detect, elf, bin, ihex, hex, tihex, srec**.
No Load Sections `target_loader_no_load_sections`StringList	Names of (loadable) sections not to load.

Simulator

Property	Description
Memory Simulation Filename `arm_simulator_memory_simulation_filename`Pro	Specifies the dll that simulates the memory system. This property will have macro expansion applied to it.
Memory Simulation Parameter `arm_simulator_memory_simulation_parameter`S	Parameter passed to the memory simulation.The format of this is specific to the memory simulation. The default memory simulation takes a list of **RX|RWX 'hex start address', 'hex size in bytes', 'default hex word value'** for example RX 00000000, 10000000, FFFFFFFF;RWX 10000000, 10000000, CDCDCDCD
Memory Simulation Parameter Macros `arm_simulator_memory_simulation_parameter_`	Macros to apply to the parameter passed to the memory simulation on creation.
Stop On Memory Error `arm_simulator_stop_on_read_write`Enumeration	Specifies the simulator behaviour when a memory error occurs.
Trace Buffer Size `arm_simulator_num_trace_entries`Integer	The number of trace entries to store.

Target Script

Property	Description
Attach Script `target_attach_script`JavaScript	The script that is executed when the target is attached to.
Connect Script `target_connect_script`JavaScript	The script that is executed when the target is connected to.
Debug Begin Script `target_debug_begin_script`JavaScript	The script that is executed when the debugger begins a debug session.
Debug End Script `target_debug_end_script`JavaScript	The script that is executed when the debugger ends a debug session.
Disconnect Script `target_disconnect_script`JavaScript	The script that is executed when the target is disconnected from.
Reset Script `target_reset_script`JavaScript	The script that is executed when the target is reset.
Target Script File `target_script_file`FileName	The target script file, the contents of this file are prepended to script project properties before they are executed.

Target Trace

Property	Description

ITM Stimulus Ports Enable `arm_target_itm_stimulus_port_enable`IntegerHe	Specifies the ITM Stimulus ports to enable.
ITM Stimulus Ports Privilege `arm_target_itm_stimulus_port_privilege`Intege	Specifies the ITM Stimulus ports to enable.
ITM Timestamping `arm_target_itm_timestamping_enable`Enumeratio	Specifies ITM timestamping. The options are: Disable - disable timestamping Local - use the local timestamp clock Global - use the global timestamp clock
ITM/DWT Data Trace PC `arm_target_dwt_data_trace_PC`Boolean	Specifies whether to trace the PC on data trace.
ITM/DWT PC Sampling `arm_target_dwt_PC_sampling_enable`Enumeration	Specifies the DWT PC sampling rate.
ITM/DWT Trace Exceptions `arm_target_dwt_trace_exceptions`Boolean	Specifies whether to trace exception entry and return.
MTB RAM Address `arm_target_mtb_ram_address`IntegerHex	Specifies the MTB RAM Address - note that this must be aligned to the MTB RAM size.
MTB RAM Size `arm_target_mtb_ram_size`Enumeration	Specifies the MTB RAM Size
SWO Baud Rate `arm_target_trace_SWO_speed`IntegerRange	The baud rate of the SWO.
Trace Clock Speed `arm_target_trace_clock_speed`IntegerRange	The speed of the trace clock. This is usually the same as the CPU clock and is used to program the prescaler for the SWO
Trace Initialize Script `target_trace_initialize_script`JavaScript	The script that is executed to initialize the target trace hardware. When executed this script has the macro $(TraceInterfaceType) expanded with value of the Trace Interface Type property, typically it is EnableTrace("$(TraceInterfaceType)").
Trace Interface Type `arm_target_trace_interface_type`Enumeration	Specifies the type of trace interface the target has. The options are: SWO - Use asynchronous SWO trace interface. TracePort - Use synchronous parallel trace interface. ETB - Use on-chip embedded trace buffer. MTB - Use on-chip MTB - Cortex-M0+ only. PC Sampling - sample the PC. None

Trace Port Size `arm_target_trace_port_size`Enumeration	Specifies the trace port size the target has. The options are: 1-bit 2-bit 4-bit 8-bit 16-bit 24-bit 32-bit

Executable Project Options

Library

Property	Description
Exclude Default Library Helper Functions `link_use_multi_threaded_libraries`Boolean	Specifies whether to exclude default library helper functions.
Include Standard Libraries `link_include_standard_libraries`Boolean	Specifies whether the standard libraries should be linked into your application.
Library Instruction Set `arm_library_instruction_set`Enumeration	Specifies the instruction set variant of the libraries to link with.
Library Optimization `arm_library_optimization`Enumeration	Specifies whether to link with libraries optimized for speed or size.
Standard Libraries Directory `link_standard_libraries_directory`String	Specifies where to find the standard libraries
Use GCC Libraries `arm_use_gcc_libraries`Boolean	Use GCC exception and RTTI libraries.

Linker

Property	Description
Additional Input Files `linker_additional_files`StringList	Enables additional object and library files to be supplied to the linker.
Additional Linker Options `linker_additional_options`StringList	Enables additional options to be supplied to the linker.
Additional Linker Options From File `linker_additional_options_from_file`ProjFileNa	Enables additional options to be supplied to the linker from a file.
Additional Output File Gap Fill Value `arm_linker_additional_output_file_gap_fill`	The value to fill gaps between sections in additional output file.
Additional Output Format `linker_output_format`Enumeration	The format used when creating an additional linked output file.The options are: **None** do not create an additional output file. **bin** create a binary file. **srec** create a Motorola S-Record file. **hex** create an Intel Hex file.
Check For Memory Segment Overflow `arm_library_check_memory_segment_overflow`B	Specifies whether the linker should check whether program sections fit in their memory segments.

DebugIO Implementation `arm_link_debugio_type`Enumeration	Specifies which DebugIO mechanism to link in. Options are **Breakpoint** (hardware breakpoint instruction and memory locations are used, not not available on v4t architecture), **DCC** (ARM debug communication channel is used), and **Memory Poll** (memory locations are polled).
Default Fill Pattern `arm_linker_script_generator_default_fill_p`	Specifies the default pattern used to fill unspecified regions of memory in a generated linker script. This pattern maybe overidden by the *fill* attribute of a program section in the section placement file.
Emit Relocations `arm_linker_emit_relocations`Boolean	Output relocation information into the executable.
Entry Point `gcc_entry_point`String	Specifies the entry point of the program.
Generate Map File `linker_map_file`Boolean	Specifies whether to generate a linkage map file.
Keep Linker Script File `keep_linker_script_file`Boolean	Keep the generated linker script file.
Keep Symbols `linker_keep_symbols`StringList	Specifies the symbols that should be kept by the linker even if they are not reachable.
Linker Script File `link_linker_script_file`ProjFileName	The name of the manual linker script file.
Linker Symbol Definitions `link_symbol_definitions`StringList	Specifies one or more linker symbol definitions.
Section Placement File `linker_section_placement_file`ProjFileName	The name of the file containing section placement description.
Section Placement Macros `linker_section_placement_macros`StringList	Macro values to substitue in section placement nodes - MACRO1=value1;MACRO2=value2.
Section Placement Segments `linker_section_placements_segments`StringList	The start and size of named segments in the section placement file, these are used when no memory map file is available.Each segment is specified by NAME RWX HEXSTART HEXSIZE for example FLASH RX 0x08000000 0x00010000
Strip Debug Information `linker_strip_debug_information`Boolean	Specifies whether debug information should be stripped from the linked image.
Strip Symbols `gcc_strip_symbols`Boolean	Specifies whether symbols should be stripped.
Suppress Warning on Mismatch `arm_linker_no_warn_on_mismatch`Boolean	No warning on mismatched object files/libraries.
Treat Linker Warnings as Errors `arm_linker_treat_warnings_as_errors`Boolean	Treat linker warnings as errors.

Use Indirect File `arm_linker_indirect_file`Boolean	Create indirect file for input files.
Use Manual Linker Script `link_use_linker_script_file`Boolean	Specifies whether to use a manual linker script.

Printf/Scanf

Property	Description
Printf Floating Point Supported `linker_printf_fp_enabled`Boolean	Are floating point numbers supported by the printf function group.
Printf Integer Support `linker_printf_fmt_level`Enumeration	The largest integer type supported by the printf function group.
Printf Width/Precision Supported `linker_printf_width_precision_supported`Bool	Enables support for width and precision specification in the printf function group.
Scanf Classes Supported `linker_scanf_character_group_matching_enab`	Enables support for %[...] and %[^...] character class matching in the scanf functions.
Scanf Floating Point Supported `linker_scanf_fp_enabled`Boolean	Are floating point numbers supported by the scanf function group.
Scanf Integer Support `linker_scanf_fmt_level`Enumeration	The largest integer type supported by the scanf function group.
Wide Characters Supported `linker_printf_wchar_enabled`Boolean	Are wide characters supported by the printf function group.

Runtime Memory Area

Property	Description
Heap Size `arm_linker_heap_size`IntegerRange	The size of the heap in bytes.
Main Stack Size `arm_linker_stack_size`IntegerRange	The size of the main stack in bytes.
Process Stack Size `arm_linker_process_stack_size`IntegerRange	The size of the process stack in bytes.
Stack Size (Abort Mode) `arm_linker_abt_stack_size`IntegerRange	The size of the Abort mode stack in bytes.
Stack Size (FIQ Mode) `arm_linker_fiq_stack_size`IntegerRange	The size of the FIQ mode stack in bytes.
Stack Size (IRQ Mode) `arm_linker_irq_stack_size`IntegerRange	The size of the IRQ mode stack in bytes.

Stack Size (Supervisor Mode) `arm_linker_svc_stack_size`IntegerRange	The size of the Supervisor mode stack in bytes.
Stack Size (Undefined Mode) `arm_linker_und_stack_size`IntegerRange	The size of the Undefined mode stack in bytes.

User Build Step

Property	**Description**
Link Patch Command `linker_patch_build_command`Unknown	A command to run after the link but prior to additional binary file generation. This property will have macro expansion applied to it with the additional **$(TargetPath)** macro set to the output filepath of the linker command.
Link Patch Working Directory `linker_patch_build_command_wd`DirPath	The working directory where the link patch command is run. This property will have macro expansion applied to it.
Post-Link Command `linker_post_build_command`Unknown	A command to run after the link command has completed.This property will have macro expansion applied to it with the additional **$(TargetPath)** macro set to the output filepath of the linker command and **$(PostLinkOutputFilePath)** set to the value of the output filepath of the post link command.
Post-Link Output File `linker_post_build_command_output_file`String	The name of the file created by the post-link command. This property will have macro expansion applied to it.
Post-Link Working Directory `linker_post_build_command_wd`DirPath	The working directory where the post-link command is run. This property will have macro expansion applied to it.
Pre-Link Command `linker_pre_build_command`Unknown	A command to run before the link command. This property will have macro expansion applied to it.
Pre-Link Working Directory `linker_pre_build_command_wd`DirPath	The working directory where the pre-link command is run. This property will have macro expansion applied to it.

System Macros

System Macro Values

Property	Description
$(Date) `$(Date)`String	Day Month Year e.g. 21 June 2011.
$(DateDay) `$(DateDay)`String	Year e.g. 2011.
$(DateMonth) `$(DateMonth)`String	Month e.g. June.
$(DateYear) `$(DateYear)`String	Day e.g. 21.
$(DesktopDir) `$(DesktopDir)`String	Path to users desktop directory.
$(DocumentsDir) `$(DocumentsDir)`String	Path to users documents directory.
$(HomeDir) `$(HomeDir)`String	Path to users home directory.
$(HostArch) `$(HostArch)`String	The CPU architecture that SEGGER Embedded Studio is running on e.g. x86.
$(HostDLL) `$(HostDLL)`String	The file extension for dynamic link libraries on the CPU that SEGGER Embedded Studio is running on e.g. .dll.
$(HostDLLExt) `$(HostDLLExt)`String	The file extension for dynamic link libraries used by the operating system that SEGGER Embedded Studio is running on e.g. .dll, .so, .dylib.
$(HostEXE) `$(HostEXE)`String	The file extension for executables on the CPU that SEGGER Embedded Studio is running on e.g. .exe.
$(HostOS) `$(HostOS)`String	The name of the operating system that SEGGER Embedded Studio is running on e.g. win.
$(Micro) `$(Micro)`String	The SEGGER Embedded Studio target e.g. ARM.
$(PackagesDir) `$(PackagesDir)`String	Path to the users packages directory.
$(Platform) `$(Platform)`String	The target platform.
$(ProductNameShort) `$(ProductNameShort)`String	The product name.
$(StudioArchiveFileExt) `$(StudioArchiveFileExt)`String	The filename extension of a studio archive file.

$(StudioBuildToolExeName) `$(StudioBuildToolExeName)`String	The filename of the build tool executable.
$(StudioBuildToolName) `$(StudioBuildToolName)`String	The name of the build tool executable.
$(StudioDir) `$(StudioDir)`String	The install directory of the product.
$(StudioExeName) `$(StudioExeName)`String	The filename of the studio executable.
$(StudioMajorVersion) `$(StudioMajorVersion)`String	The major release version of software.
$(StudioMinorVersion) `$(StudioMinorVersion)`String	The minor release version of software.
$(StudioName) `$(StudioName)`String	The full name of studio.
$(StudioNameShort) `$(StudioNameShort)`String	The short name of studio.
$(StudioPackageFileExt) `$(StudioPackageFileExt)`String	The filename extension of a studio package file.
$(StudioProjectFileExt) `$(StudioProjectFileExt)`String	The filename extension of a studio project file.
$(StudioScriptToolExeName) `$(StudioScriptToolExeName)`String	The filename of the script tool executable.
$(StudioScriptToolName) `$(StudioScriptToolName)`String	The name of the script tool executable.
$(StudioSessionFileExt) `$(StudioSessionFileExt)`String	The filename extension of a studio session file.
$(StudioUserDir) `$(StudioUserDir)`String	The directory containing the user data.
$(TargetID) `$(TargetID)`String	ID number representing the SEGGER Embedded Studio target.
$(Time) `$(Time)`String	Hour:Minutes:Seconds e.g. 15:34:03.
$(TimeHour) `$(TimeHour)`String	Hour e.g. 15.
$(TimeMinute) `$(TimeMinute)`String	Hour e.g. 34.
$(TimeSecond) `$(TimeSecond)`String	Hour e.g. 03.

Build Macros

(Build Macro Values)

Property	Description
$(Arch) `$(Arch)`String	The lower case value of the **ARM Architecture** project property.
$(AsmOptions) `$(AsmOptions)`String	A space seperated list of assembler options for the external assemble command.
$(COnlyOptions) `$(COnlyOptions)`String	A space seperated list of compiler options for the external c compile command.
$(COptions) `$(COptions)`String	A space seperated list of compiler options for the external c and c++ compile commands.
$(CombiningOutputFilePath) `$(CombiningOutputFilePath)`String	The full path of the output file of the combining command.
$(CombiningRelInputPaths) `$(CombiningRelInputPaths)`String	The relative inputs to the combining command.
$(Configuration) `$(Configuration)`String	The build configuration e.g. ARM Flash Debug.
$(CoreType) `$(CoreType)`String	The lower case value of the **ARM Core Type** project property.
$(Defines) `$(Defines)`String	The preprocessor defines property value for the external compile command.
$(DependencyPath) `$(DependencyPath)`String	The path of the dependency file for the external compile command.
$(EXE) `$(EXE)`String	The default file extension for an executable file including the dot e.g. .elf.
$(Endian) `$(Endian)`String	The lower case value of the **Byte Order** project property.
$(FPU) `$(FPU)`String	The lower case value of the **ARM FPU Type** project property.
$(FPU2) `$(FPU2)`String	Alternative value of the **ARM FPU Type** project property.
$(FPU3) `$(FPU3)`String	Alternative value of the **ARM FPU Type** project property.
$(FolderName) `$(FolderName)`String	The folder name of the containing folder.
$(GCCTarget) `$(GCCTarget)`String	The value of the **GCC Target** project property.

$(Includes) `$(Includes)`String	The user includes property value for the external compile command.
$(InputDir) `$(InputDir)`String	The absolute directory of the input file.
$(InputExt) `$(InputExt)`String	The extension of an input file not including the dot e.g cpp.
$(InputFileName) `$(InputFileName)`String	The name of an input file relative to the project directory.
$(InputName) `$(InputName)`String	The name of an input file relative to the project directory without the extension.
$(InputPath) `$(InputPath)`String	The absolute name of an input file including the extension.
$(IntDir) `$(IntDir)`String	The macro-expanded value of the **Intermediate Directory** project property.
$(LIB) `$(LIB)`String	The default file extension for a library file including the dot e.g. .lib.
$(LibExt) `$(LibExt)`String	The architecture and build specific library extension.
$(LinkOptions) `$(LinkOptions)`String	A space seperated list of compiler options for the external link command.
$(LinkerScriptPath) `$(LinkerScriptPath)`String	The full path of the linker script file for the link command.
$(MapPath) `$(MapPath)`String	The full path of the map file of the external link command.
$(OBJ) `$(OBJ)`String	The default file extension for an object file including the dot e.g. .o.
$(Objects) `$(Objects)`String	A space seperated list of files for the external archive or link command.
$(ObjectsFilePath) `$(ObjectsFilePath)`String	The filename containing the files for the external archive or link command.
$(OutDir) `$(OutDir)`String	The macro-expanded value of the **Output Directory** project property.
$(PackageExt) `$(PackageExt)`String	The file extension of a package file e.g. emPackage.
$(PostLinkOutputFilePath) `$(PostLinkOutputFilePath)`String	The full path of the output file of the post link command.
$(ProjectDir) `$(ProjectDir)`String	The absolute value of the **Project Directory** project property of the current project. If this isn't set then the directory containing the solution file.
$(ProjectName) `$(ProjectName)`String	The project name of the current project.

$(ProjectNodeName) `$(ProjectNodeName)`String	The name of the selected project node.
$(RelInputPath) `$(RelInputPath)`String	The relative path of the input file to the project directory.
$(RelTargetPath) `$(RelTargetPath)`String	The project directory relative path of the output file of the link or compile command.
$(RootOutDir) `$(RootOutDir)`String	The macro-expanded value of the **Root Output Directory** project property.
$(RootRelativeOutDir) `$(RootRelativeOutDir)`String	The relative path to get from the path specified by the **Output Directory** project property to the path specified by the **Root Output Directory** project property.
$(SolutionDir) `$(SolutionDir)`String	The absolute path of the directory containing the solution file.
$(SolutionExt) `$(SolutionExt)`String	The extension of the solution file without the dot.
$(SolutionFileName) `$(SolutionFileName)`String	The filename of the solution file.
$(SolutionName) `$(SolutionName)`String	The basename of the solution file.
$(SolutionPath) `$(SolutionPath)`String	The absolute path of the solution file.
$(StageOutputFilePath) `$(StageOutputFilePath)`String	The full path of the output file of the stage command.
$(TargetPath) `$(TargetPath)`String	The full path of the output file of the link or compile command.
$(ToolChainDir) `$(ToolChainDir)`String	The macro-expanded value of the **Tool Chain Directory** project property.

BinaryFile

The following table lists the BinaryFile object's member functions.

BinaryFile.crc32(offset, length) returns the CRC-32 checksum of an address range *length* bytes long, starting at *offset*. This function computes a CRC-32 checksum on a block of data using the standard CRC-32 polynomial (0x04C11DB7) with an initial value of 0xFFFFFFFF. Note that this implementation doesn't reflect the input or the output and the result is inverted.
BinaryFile.length() returns the length of the binary file in bytes.
BinaryFile.load(path) loads binary file from *path*.
BinaryFile.peekBytes(offset, length) returns byte array containing *length* bytes peeked from *offset*.
BinaryFile.peekUint32(offset, littleEndian) returns a 32-bit word peeked from *offset*. The *littleEndian* argument specifies the endianness of the access, if true or undefined it will be little endian, otherwise it will be big endian.
BinaryFile.pokeBytes(offset, byteArray) poke byte array *byteArray* to *offset*.
BinaryFile.pokeUint32(offset, value, littleEndian) poke a *value* to 32-bit word located at *offset*. The *littleEndian* argument specifies the endianness of the access, if true or undefined it will be little endian, otherwise it will be big endian.
BinaryFile.resize(length, fill) resizes the binary image to *length* bytes. If the operation extends the size, the binary image will be padded with bytes of value *fill*.
BinaryFile.save(path) saves binary file to *path*.
BinaryFile.saveRange(path, offset, length) saves part of the binary file to *path*. The *offset* argument specifies the byte offset to start from. The *length* argument specifies the maximum number of bytes that should be saved.

CWSys

The following table lists the CWSys object's member functions.

CWSys.appendStringToFile(path, string) appends *string* to the end of the file *path*.
CWSys.copyFile(srcPath, destPath) copies file *srcPath* to *destPath*.
CWSys.crc32(array) returns the CRC-32 checksum of the byte array *array*. This function computes a CRC-32 checksum on a block of data using the standard CRC-32 polynomial (0x04C11DB7) with an initial value of 0xFFFFFFFF. Note that this implementation doesn't reflect the input or the output and the result is inverted.
CWSys.fileExists(path) returns true if file *path* exists.
CWSys.fileSize(path) return the number of bytes in file *path*.
CWSys.getRunStderr() returns the stderr output from the last *CWSys.run()* call.
CWSys.getRunStdout() returns the stdout output from the last *CWSys.run()* call.
CWSys.makeDirectory(path) create the directory *path*.
CWSys.packU32(array, offset, number, le) packs *number* into the *array* at *offset*.
CWSys.popup(text) prompt the user with text and return true for yes and false for no.
CWSys.readByteArrayFromFile(path) returns the byte array contained in the file *path*.
CWSys.readStringFromFile(path) returns the string contained in the file *path*.
CWSys.removeDirectory(path) remove the directory *path*.
CWSys.removeFile(path) deletes file *path*.
CWSys.renameFile(oldPath, newPath) renames file *oldPath* to be *newPath*.
CWSys.run(cmd, wait) runs command line *cmd* optionally waits for it to complete if *wait* is true.
CWSys.unpackU32(array, offset, le) returns the number unpacked from the *array* at *offset*.
CWSys.writeByteArrayToFile(path, array) creates a file *path* containing the byte array *array*.
CWSys.writeStringToFile(path, string) creates a file *path* containing *string*.

Debug

The following table lists the Debug object's member functions.

Debug.breakexpr(expression, count, hardware) set a breakpoint on *expression*, with optional ignore *count* and use *hardware* parameters. Return the, none zero, allocated breakpoint number.
Debug.breakline(filename, linenumber, temporary, count, hardware) set a breakpoint on *filename* and *linenumber*, with optional *temporary*, ignore *count* and use *hardware* parameters. Return the, none zero, allocated breakpoint number.
Debug.breaknow() break execution now.
Debug.deletebreak(number) delete the specified breakpoint or all breakpoints if zero is supplied.
Debug.disassembly(source, labels, before, after) set debugger mode to disassembly mode. Optionally specify *source* and *labels* to be displayed and the number of bytes to disassemble *before* and *after* the located program counter.
Debug.echo(s) display string.
Debug.enableexception(exception, enable) *enable* break on *exception*.
Debug.evaluate(expression) evaluates debug *expression* and returns it as a JavaScript value.
Debug.getfilename() return located filename.
Debug.getlineumber() return located linenumber.
Debug.go() continue execution.
Debug.locate(frame) locate the debugger to the optional *frame* context.
Debug.locatepc(pc) locate the debugger to the specified *pc*.
Debug.locateregisters(registers) locate the debugger to the specified *register* context.
Debug.print(expression, fmt) evaluate and display debug*expression* using optional *fmt*. Supported formats are *b* binary, *c* character, *d* decimal, *e* scientific float, *f* decimal float, *g* scientific or decimal float, *i* signed decimal, *o* octal, *p* pointer value, *s* null terminated string, *u* unsigned decimal, *x* hexadecimal.
Debug.printglobals() display global variables.
Debug.printlocals() display local variables.
Debug.quit() stop debugging.
Debug.setprintarray(elements) set the maximum number of array elements for printing variables.
Debug.setprintradix(radix) set the default radix for printing variables.
Debug.setprintstring(c) set the default to print character pointers as strings.
Debug.showbreak(number) show information on the specified breakpoint or all breakpoints if zero is supplied.
Debug.showexceptions() show the exceptions.
Debug.source(before, after) set debugger mode to source mode. Optionally specify the number of source lines to display *before* and *after* the location.
Debug.stepinto() step an instruction or a statement.

Debug.stepout() continue execution and break on return from current function.	
Debug.stepover() step an instruction or a statement stepping over function calls.	
Debug.stopped() return stopped state.	
Debug.wait(ms) wait *ms* millseconds for a breakpoint and return the number of the breakpoint that hit.	
Debug.where() display call stack.	

ElfFile

The following table lists the ElfFile object's member functions.

ElfFile.crc32(address, length, virtualNotPhysical, padding) returns the CRC-32 checksum of an address range *length* bytes long, located at *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address. If *padding* is defined, it specifies the byte value used to fill gaps in the program. This function computes a CRC-32 checksum on a block of data using the standard CRC-32 polynomial (0x04C11DB7) with an initial value of 0xFFFFFFFF. Note that this implementation doesn't reflect the input or the output and the result is inverted.
ElfFile.findProgram(address) returns an object with *start*, the *data* and the *size* to allocate of the Elf program that contains *address*.
ElfFile.getEntryPoint() returns the entry point in the ELF file.
ElfFile.getSection(name) returns an object with *start* and the *data* of the Elf section corresponding to the *name*.
ElfFile.isLittleEndian() returns true if the Elf file has numbers encoded as little endian.
ElfFile.load(path) loads Elf file from *path*.
ElfFile.peekBytes(address, length, virtualNotPhysical, padding) returns byte array containing *length* bytes peeked from *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address. If *padding* is defined, it specifies the byte value used to fill gaps in the program.
ElfFile.peekUint32(address, virtualNotPhysical) returns a 32-bit word peeked from *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address.
ElfFile.pokeBytes(address, byteArray, virtualNotPhysical) poke byte array *byteArray* to *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address.
ElfFile.pokeUint32(address, value, virtualNotPhysical) poke a *value* to 32-bit word located at *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address.
ElfFile.save(path) saves Elf file to *path*.
ElfFile.symbolValue(symbol) returns the value of *symbol* in Elf file.

TargetInterface

The following table lists the TargetInterface object's member functions.

TargetInterface.beginDebugAccess() puts the target into debug state if it is not already in order to carry out a number of debug operations. The idea behind beginDebugAccess and endDebugAccess is to minimize the number of times the target enters and exits debug state when carrying out a number of debug operations. Target interface functions that require the target to be in debug state (such as peek and poke) also use beginDebugAccess and endDebugAccess to get the target into the correct state. A nesting count is maintained, incremented by beginDebugAccess and decremented by endDebugAccess. The initial processor state is recorded on the first nested call to beginDebugAccess and this state is restored when the final endDebugAccess is called causing the count to return to it initial state.
TargetInterface.commReadWord() returns a word from the ARM7/ARM9 debug comms channel.
TargetInterface.commWriteWord(word) writes a word to the ARM7/ARM9 debug comms channel.
TargetInterface.crc32(address, length) reads a block of bytes from target memory starting at address for length bytes, generates a crc32 on the block of bytes and returns it.
TargetInterface.cycleTCK(n) provide n TCK clock cycles.
TargetInterface.delay(ms) waits for ms milliseconds
TargetInterface.downloadDebugHandler() downloads the debug handler as specified by the Debug Handler File Path/Load Address project properties and uses the debug handler for the target connection.
TargetInterface.endDebugAccess(alwaysRun) restores the target run state recorded at the first nested call to beginDebugAccess. See beginDebugAccess for more information. If alwaysRun is non-zero the processor will exit debug state on the last nested call to endDebugAccess.
TargetInterface.eraseBytes(address,length) erases a length block of target memory starting at address.
TargetInterface.error(message) terminates execution of the script and outputs a target interface error message to the target log.
TargetInterface.executeFunction(address, parameter, timeout) calls a function at address with the parameter and returns the function result. The timeout is in milliseconds.
TargetInterface.executeMCR(opcode) interprets/executes the opcode assuming it to be an MRC instruction and returns the value of the specified coprocessor register.
TargetInterface.executeMCR(opcode, value) interprets/executes the opcode assuming it to be an MCR instruction that writes value to the specified coprocessor register.
TargetInterface.expandMacro(string) returns the string with macros expanded.
TargetInterface.fillScanChain(bool, lsb, msb) sets bits from lsb (least significant bit) to msb (most significant bit) in internal buffer to bool value.
TargetInterface.getDebugRegister(address) returns the value of the ADIv5 debug register denoted by address. Address has the nibble sized access point number starting at bit 24 and the register number in the bottom byte.
TargetInterface.getICEBreakerRegister(r) returns the value of the ARM7/ARM9/ARM11/CortexA/CortexR debug register r.
TargetInterface.getProjectProperty(savename) returns the value of the savename project property.

TargetInterface.getRegister(registername) returns the value of the register, register is a string specifying the register to get and must be one of r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, sp, lr, pc, cpsr, r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, r13_fiq, r14_fiq, spsr_fiq, r13_svc, r14_svc, spsr_svc, r13_abt, r14_abt, spsr_abt, r13_irq, r14_irq, spsr_irq, r13_und, r14_und, spsr_und.

TargetInterface.getTDO() return the TDO signal.

TargetInterface.getTargetProperty(savename) returns the value of the savename target property.

TargetInterface.go() allows the target to run.

TargetInterface.idcode() returns the JTAG idcode of the target.

TargetInterface.implementation() returns a string defining the target interface implementation.

TargetInterface.isStopped() returns true if the target is stopped.

TargetInterface.message(message) outputs a target interface message to the target log.

TargetInterface.packScanChain(data, lsb, msb) packs data from lsb (least significant bit) to msb (most significant bit) into internal buffer.

TargetInterface.peekBinary(address, length, filename) reads a block of bytes from target memory starting at address for length bytes and writes them to filename.

TargetInterface.peekByte(address) reads a byte of target memory from address and returns it.

TargetInterface.peekBytes(address, length) reads a block of bytes from target memory starting at address for length bytes and returns the result as an array containing the bytes read.

TargetInterface.peekMultUint16(address, length) reads length unsigned 16-bit integers from target memory starting at address and returns them as an array.

TargetInterface.peekMultUint32(address, length) reads length unsigned 32-bit integers from target memory starting at address and returns them as an array.

TargetInterface.peekUint16(address) reads a 16-bit unsigned integer from target memory from address and returns it.

TargetInterface.peekUint32(address) reads a 32-bit unsigned integer from target memory from address and returns it.

TargetInterface.peekWord(address) reads a word as an unsigned integer from target memory from address and returns it.

TargetInterface.pokeBinary(address, filename) reads a block of bytes from filename and writes them to target memory starting at address.

TargetInterface.pokeByte(address, data) writes the byte data to address in target memory.

TargetInterface.pokeBytes(address, data) writes the array data containing 8-bit data to target memory at address.

TargetInterface.pokeMultUint16(address, data) writes the array data containing 16-bit data to target memory at address.

TargetInterface.pokeMultUint32(address, data) writes the array data containing 32-bit data to target memory at address.

TargetInterface.pokeUint16(address, data) writes data as a 16-bit value to address in target memory.

TargetInterface.pokeUint32(address, data) writes data as a 32-bit value to address in target memory.

TargetInterface.pokeWord(address, data) writes data as a word value to address in target memory.
TargetInterface.readBinary(filename) reads a block of bytes from filename and returns them in an array.
TargetInterface.reset() resets the target, optionally executes the reset script and lets the target run.
TargetInterface.resetAndStop(delay) resets the target by cycling nSRST and then stops the target. delay is the number of milliseconds to hold the target in reset.
TargetInterface.resetAndStopAtZero(delay) sets a breakpoint on the instruction at address zero execution, resets the target by cycling nSRST and waits for the breakpoint to be hit. delay is the number of milliseconds to hold the target in reset.
TargetInterface.resetDebugInterface() resets the target interface (not the target).
TargetInterface.runFromAddress(address, timeout) start the target executing at address and waits for a breakpoint to be hit. The timeout is in milliseconds.
TargetInterface.runFromToAddress(from, to, timeout) start the target executing at address from and waits for the breakpoint to be hit. The timeout is in milliseconds.
TargetInterface.runTestIdle() moves the target JTAG state machine into Run-Test/Idle state
TargetInterface.runToAddress(address, timeout) sets a breakpoint at address, starts the target executing and waits for the breakpoint to be hit. The timeout is in milliseconds.
TargetInterface.scanDR(length, count) scans length bits from the internal buffer into the data register and puts the result into the internal buffer (count specifies the number of times the function is done).
TargetInterface.scanIR(length, count) scans length bits from the internal buffer into the instruction register and puts the result into the internal buffer (count specifies the number of times the function is done).
TargetInterface.selectDevice(irPre, irPost, drPre, drPost) sets the instruction and data register (number of devices) pre and post bits.
TargetInterface.setDBGRQ(v) sets/clears the DBGRQ bit of the ARM7/ARM9 debug control register.

TargetInterface.setDebugInterfaceProperty("reset_debug_interface_enabled", bool) turn on/off the reset of the debug interface.
TargetInterface.setDebugInterfaceProperty("has_etm", bool) set the ARM7/ARM9 property to enable use of the ETM.
TargetInterface.setDebugInterfaceProperty("reset_delay", N) set the XScale reset delay property to N.
TargetInterface.setDebugInterfaceProperty("post_reset_delay", N) set the XScale post reset delay property to N.
TargetInterface.setDebugInterfaceProperty("post_reset_cycles", N) set the XScale post reset cycles property to N.
TargetInterface.setDebugInterfaceProperty("post_ldic_cycles", N) set the XScale ldic cycles property to N.
TargetInterface.setDebugInterfaceProperty("sync_exception_vectors", bool) turn on/off the XScale sync exception vectors property.
TargetInterface.setDebugInterfaceProperty("peek_flash_workaround", bool) turn on/off the ARMv6M/ARMv7M peek flash memory workaround debug property.
TargetInterface.setDebugInterfaceProperty("adiv5_fast_delay_cycles", N) set the ADIv5 fast delay cycles property to N (FTDI2232 target interfaces only).
TargetInterface.setDebugInterfaceProperty("use_adiv5_AHB", N, [start, size]) set the ARMv7A/ARMv7R debug property list to turn on/off usage of the ADIv5 AHB MEM-AP for 1+2+4 data sized accesses on the optional address range specified by start and size.
TargetInterface.setDebugInterfaceProperty("set_adiv5_AHB_ap_num", N) specify the ARMv6M/ARMv7A/ARMv7M/ARMv7R AHB AP number to use.
TargetInterface.setDebugInterfaceProperty("set_adiv5_APB_ap_num", N) specify the ARMv7A/ARMv7R APB AP number to use.
TargetInterface.setDebugInterfaceProperty("max_ap_num", N) set the ADIv5 debug property to limit the number of AP's to detect to N.
TargetInterface.setDebugInterfaceProperty("component_base", N) set the ADIv5 debug property that specifies the base address N of the CoreSight debug component.

TargetInterface.setDebugRegister(address, value) set the value of the ADIv5 debug register denoted by address. Address has the nibble sized access point number starting at bit 24 and the register number in the bottom byte.

TargetInterface.setDeviceTypeProperty(type) sets the target interface's Device Type property string to type. This would typically be used by a Connect Script to override the default Device Type property and provide a custom description of the connected target.

TargetInterface.setICEBreakerBreakpoint(n, address, addressMask, data, dataMask, control, controlMask) sets the ARM7/ARM9 watchpoint n registers.

TargetInterface.setICEBreakerRegister(r, value) set the value of the ARM7/ARM9/ARM11/CortexA/CortexR debug register r.

TargetInterface.setMaximumJTAGFrequency(hz) allows the maximum TCK frequency of the currently connected JTAG interface to be set dynamically. The speed setting will only apply for the current connection session, if you reconnect the setting will revert to the speed specfied by the target interface properties. Calls to this function will be ignored if adaptive clocking is being used.

TargetInterface.setNSRST(v) sets/clears the NSRST signal.

TargetInterface.setNTRST(v) sets/clears the NTRST signal.

TargetInterface.setRegister(registername, value) sets the register to the value, register is a string specifying the register to get and must be one of r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, sp, lr, pc, cpsr, r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, r13_fiq, r14_fiq, spsr_fiq, r13_svc, r14_svc, spsr_svc, r13_abt, r14_abt, spsr_abt, r13_irq, r14_irq, spsr_irq, r13_und, r14_und, spsr_und.
TargetInterface.setTDI(v) clear/set TDI signal.
TargetInterface.setTMS(v) clear/set TMS signal.
TargetInterface.setTargetProperty(savename) set the value of the savename target property.
TargetInterface.stop() stops the target.
TargetInterface.stopAndReset(delay) sets a breakpoint on any instruction execution, resets the target by cycling nSRST and waits for the breakpoint to be hit. delay is the number of milliseconds to hold the device in reset.
TargetInterface.trst() resets the target interface (not the target).
TargetInterface.type() returns a string defining the target interface type.
TargetInterface.unpackScanChain(lsb, msb) unpacks data from lsb (least significant bit) to msb (most significant bit) from internal buffer and returns the result.
TargetInterface.waitForDebugState(timeout) waits for the target to stop or the timeout in milliseconds.
TargetInterface.writeBinary(array, filename) write the bytes in array to filename.

WScript

The following table lists the WScript object's member functions.

WScript.Echo(s)	echos string *s* to the output terminal.

X-ON Electronics

Largest Supplier of Electrical and Electronic Components

Click to view similar products for Development Software ***category:***

Click to view products by Segger Microcontroller ***manufacturer:***

Other Similar products are found below :

RAPPID-567XFSW SRP004001-01 SW163052 SYSWINEV21 WS01NCTF1E W128E13 SW89CN0-ZCC IPS-EMBEDDED IP-UART-16550 MPROG-PRO535E AFLCF-08-LX-CE060-R21 WS02-CFSC1-EV3-UP SYSMAC-STUDIO-EIPCPLR LIB-PL-PC-N-1YR-DISKID 1120270005 1120270006 MIKROBASIC PRO FOR FT90X (USB DONGLE) MIKROC PRO FOR FT90X (USB DONGLE) MIKROC PRO FOR PIC (USB DONGLE LICENSE) MIKROBASIC PRO FOR AVR (USB DONGLE LICEN MIKROBASIC PRO FOR FT90X MIKROC PRO FOR DSPIC30/33 (USB DONGLE LI MIKROPASCAL PRO FOR ARM (USB DONGLE LICE MIKROPASCAL PRO FOR FT90X MIKROPASCAL PRO FOR FT90X (USB DONGLE) MIKROPASCAL PRO FOR PIC32 (USB DONGLE LI SW006021-2H ATATMELSTUDIO 2400573 2702579 2988609 2702546 SW006022-DGL 2400303 2701356 VDSP-21XX-PCFLOAT VDSP-BLKFN-PC-FULL 88970111 DG-ACC-NET-CD 55195101-101 55195101-102 SW1A-W1C MDK-ARM SW006021-2NH SW006021-1H SW006021-2 SW006022-2 SW006023-2 SW007023 MIKROE-730