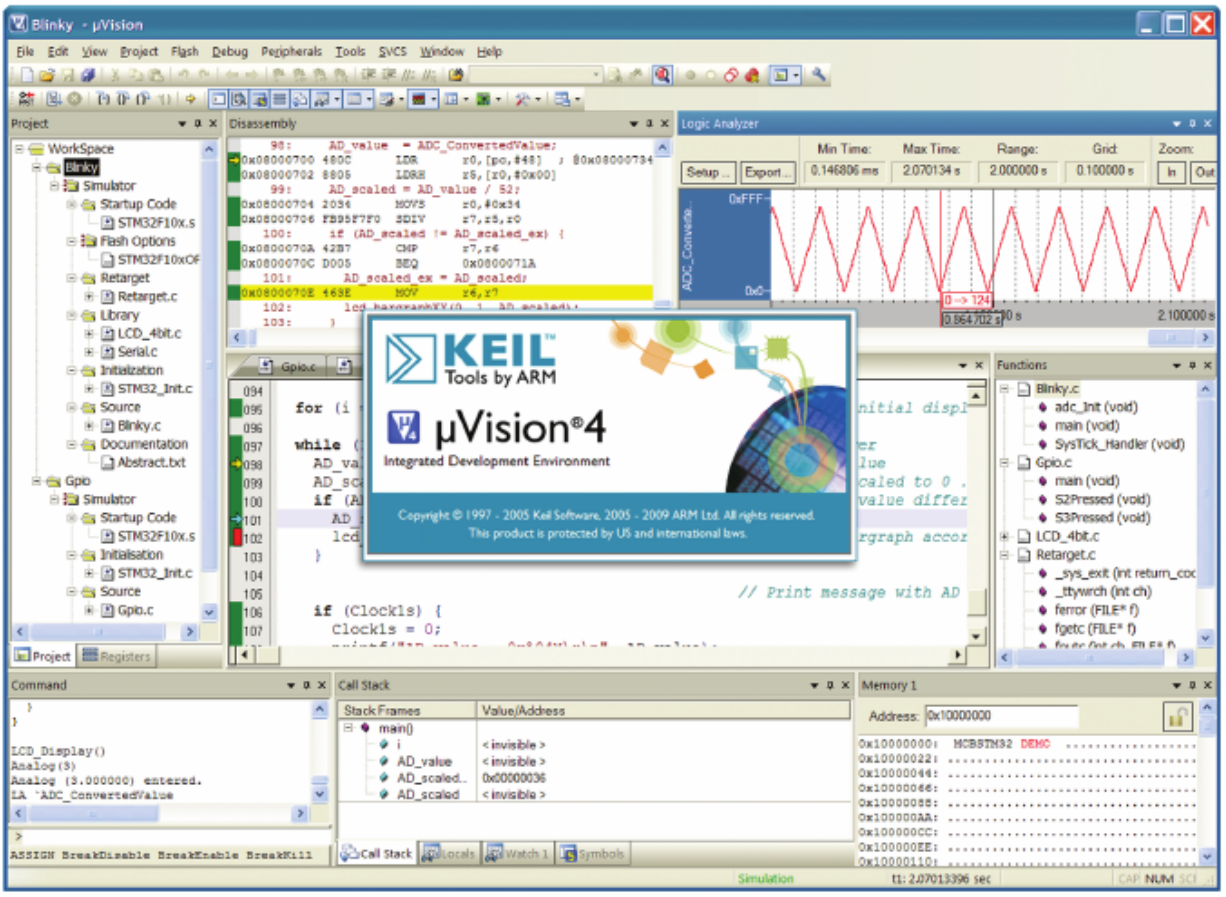
KEIL™
Tools by ARM

Getting Started

Creating Applications with µVision®4



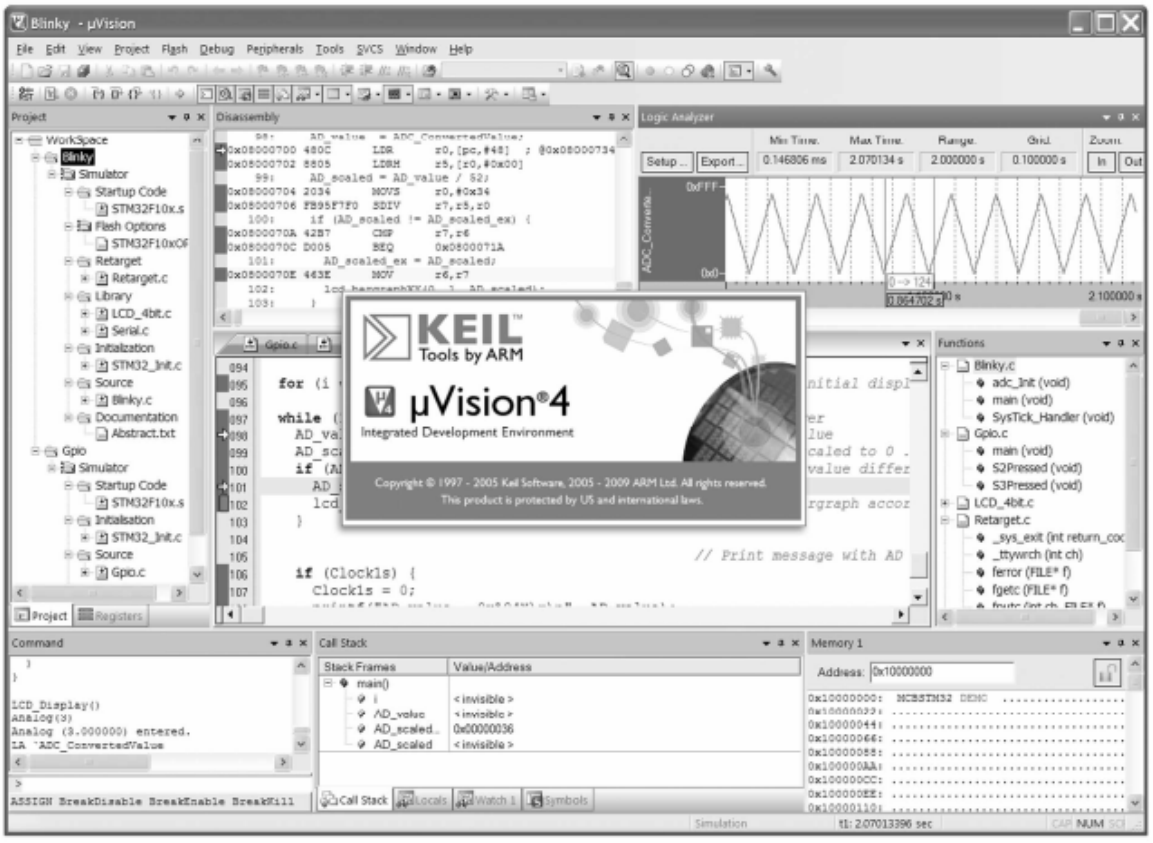
For 8-bit, 16-bit, and 32-bit Microcontrollers

www.keil.com

KEIL™
Tools by ARM

Getting Started

Creating Applications with µVision®4



For 8-bit, 16-bit, and 32-bit Microcontrollers

NOTE
This manual assumes that you are familiar with Microsoft Windows and the hardware and instruction set of the ARM7, ARM9, Cortex-Mx, C166, XE166, XC2000, or 8051 microcontroller.

Every effort was made to ensure accuracy in this manual and to give appropriate credit to persons, companies, and trademarks referenced herein.

Preface

This manual is an introduction to the Keil development tools designed for Cortex-Mx, ARM7, ARM9, C166, XE166, XC2000, and 8051 microcontrollers. It introduces the µVision Integrated Development Environment, Simulator, and Debugger and presents a step-by-step guided tour of the numerous features and capabilities the Keil embedded development tools offer.

Who should Read this Book

This book is useful for students, beginners, advanced and experienced developers alike.

Developers are considered experienced or advanced if they have used µVision extensively in the past and knowledge exists of how the µVision IDE works and interacts with the debugger, simulator, and target hardware. Preferably, these developers already have a deep understanding of microcontrollers. We encourage this group of engineers to get familiar with the enhancements introduced and to explore the latest features in µVision.

Developers are considered students or beginners if they have no working experience with µVision. We encourage this group of developers to start by reading the chapters related to the µVision IDE and to work through the examples to get familiar with the interface and configuration options described. They should make use of the ample possibilities the simulator offers. Later on, they should continue with the chapters describing the RTOS and microcontroller architectures.

However, it is assumed that you have a basic knowledge of how to use microcontrollers and that you are familiar with a few instructions or with the instruction set of your preferred microcontroller.

The chapters of this book can be studied individually, since they do not strictly depend on each other.

Chapter Overview

"Chapter 1. **Introduction**", provides an overview of product installation and licensing and shows how to get support for the Keil development tools.

"Chapter 2. **Microcontroller Architectures**", discusses various microcontroller architectures supported by the Keil development tools and assists you in choosing the microcontroller best suited for your application.

"Chapter 3. **Development Tools**", discusses the major features of the µVision IDE and Debugger, Assembler, Compiler, Linker, and other development tools.

"Chapter 4. **RTX RTOS Kernel**", discusses the benefits of using a Real-Time Operating System (RTOS) and introduces the features available in Keil RTX Kernels.

"Chapter 5. **Using µVision**", describes specific features of the µVision user interface and how to interact with them.

"Chapter 6. **Creating Embedded Programs**", describes how to create projects, edit source files, compile, fix syntax errors, and generate executable code.

"Chapter 7. **Debugging**", describes how to use the µVision Simulator and Target Debugger to test and validate your embedded programs.

"Chapter 8. **Using Target Hardware**", describes how to configure and use third-party Flash programming utilities and target drivers.

"Chapter 9. **Example Programs**", describes four example programs and shows the relevant features of µVision by means of these examples.

Document Conventions

Examples	Description
README.TXT[1]	Bold capital text is used to highlight the names of executable programs, data files, source files, environment variables, and commands that you can enter at the command prompt. This text usually represents commands that you must type in literally. For example: **ARMCC.EXE DIR LX51.EXE**
`Courier`	Text in this typeface is used to represent information that is displayed on the screen or is printed out on the printer This typeface is also used within the text when discussing or describing command line items.
Variables	Text in italics represents required information that you must provide. For example, *projectfile* in a syntax string means that you must supply the actual project file name Occasionally, italics are also used to emphasize words in the text.
Elements that repeat…	Ellipses (…) are used to indicate an item that may be repeated

Omitted code
.
.
.
	Vertical ellipses are used in source code listings to indicate that a fragment of the program has been omitted. For example:
	void main (void) {
	.
	.
	.
	while (1);

«Optional Items»	Double brackets indicate optional items in command lines and input fields. For example: **C51 TEST.C PRINT** «filename»
{ opt1 | opt2 }	Text contained within braces, separated by a vertical bar represents a selection of items. The braces enclose all of the choices and the vertical bars separate the choices. Exactly one item in the list must be selected.
Keys	Text in this sans serif typeface represents actual keys on the keyboard. For example, "Press Enter to continue

[1] *It is not required to enter commands using all capital letters.*

Contents

Chapter 1. Introduction

Thank you for allowing Keil to provide you with software development tools for your embedded microcontroller applications.

This book, **Getting Started**, describes the µVision IDE, µVision Debugger and Analysis Tools, the simulation, and debugging and tracing capabilities. In addition to describing the basic behavior and basic screens of µVision, this book provides a comprehensive overview of the supported microcontroller architecture types, their advantages and highlights, and supports you in selecting the appropriate target device. This book incorporates hints to help you to write better code. As with any **Getting Started** book, it does not cover every aspect and the many available configuration options in detail. We encourage you to work through the examples to get familiar with µVision and the components delivered.

The Keil Development Tools are designed for the professional software developer, however programmers of all levels can use them to get the most out of the embedded microcontroller architectures that are supported.

Tools developed by Keil endorse the most popular microcontrollers and are distributed in several packages and configurations, dependent on the architecture.

- **MDK-ARM**: Microcontroller Development Kit, for several ARM7, ARM9, and Cortex-Mx based devices
- **PK166**: Keil Professional Developer's Kit, for C166, XE166, and XC2000 devices
- **DK251**: Keil 251 Development Tools, for 251 devices
- **PK51**: Keil 8051 Development Tools, for Classic & Extended 8051 devices

In addition to the software packages, Keil offers a variety of evaluation boards, USB-JTAG adapters, emulators, and third-party tools, which completes the range of products.

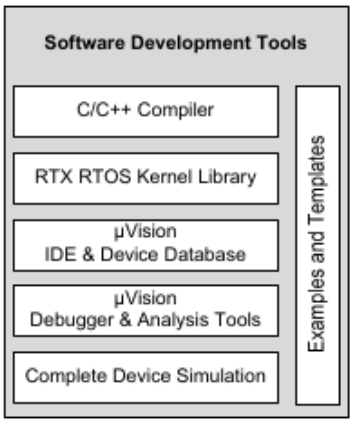
The following illustrations show the generic component blocks of µVision in conjunction with tools provided by Keil, or tools from other vendors, and the way the components relate.

Software Development Tools

Like all software based on Keil's µVision IDE, the toolsets provide a powerful, easy to use and easy to learn environment for developing embedded applications.

They include the components you need to create, debug, and assemble your C/C++ source files, and incorporate simulation for microcontrollers and related peripherals.

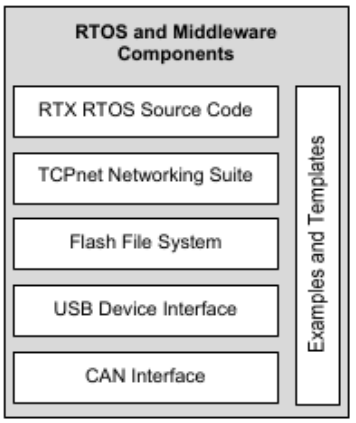
The RTX RTOS Kernel helps you to implement complex and time-critical software.



RTOS and Middleware Components

These components are designed to solve communication and real-time challenges of embedded systems. While it is possible to implement embedded applications without using a real-time kernel, a proven kernel saves time and shortens the development cycle.

This component also includes the source code files for the operating system.



Hardware Debug Adapters

The µVision Debugger fully supports several emulators provided by Keil, and other vendors. The Keil ULINK USB-JTAG family of adapters con nect the USB port of a PC to the target hardware. They enable you to download, test, and debug your embedded application on real hardware.

Last-Minute Changes

As with any high-tech product, last minute changes might not be included into the printed manuals. These last-minute changes and enhancements to the software and manuals are listed in the **Release Notes** shipped with the product.

Licensing

Each Keil product requires activation through a license code. This code is obtained via e-mail during the registration process. There are two types of product licenses:

- **Single-User License** is available for all Keil products. A Single-User License grants the right to use a product on a maximum of two computers to one user. Each installation requires a license code that is personalized for the computer on which the product is installed. A Single-User license may be uninstalled and moved to another computer.
- **Floating-User License** is available for many Keil products. The Floating-User license grants the right to use that product on several computers by several different developers at the same time. Each installation of the product requires an individual license code for each computer on which the product is installed.

Installation

Please check the minimum hardware and software requirements that must be satisfied to ensure that your Keil development tools are installed and will function properly. Before attempting installation, verify that you have:

- A standard PC running Microsoft Windows XP, or Windows Vista
- 1GB RAM and 500 MB of available hard-disk space is recommended
- 1024x768 or higher screen resolution; a mouse or other pointing device
- A CD-ROM drive

Keil products are available on CD-ROM and via download from **www.keil.com**. Updates to the related products are regularly available at **www.keil.com/update**.

Installation using the web download

1. Download the product from **www.keil.com/demo**
2. Run the downloaded executable
3. Follow the instructions displayed by the **SETUP** program

Installation from CD-ROM

1. Insert the CD-ROM into your CD-ROM drive. The CD-ROM browser should start automatically. If it does not, you can run **SETUP.EXE** from the CD-ROM.
2. Select **Install Products & Updates** from the CD Browser menu
3. Follow the instructions displayed by the **SETUP** program

Product Folder Structure

The **SETUP** program copies the development tools into subfolders. The base folder defaults to **C:\KEIL**. The following table lists the default folders for each microcontroller architecture installation. Adjust the examples used in this manual to your preferred installation directory accordingly.

Microcontroller Architecture	Folder
MDK-ARM Toolset	**C:\KEIL\ARM**
C166/XE166/XC2000 Toolset	**C:\KEIL\C166**
8051 Toolset	**C:\KEIL\C51**
C251 Toolset	**C:\KEIL\C251**
µVision Common Files	**C:\KEIL\UV4**

Each toolset contains several subfolders:

Contents	Subfolder
Executable Program Files	**\BIN**
C Include/Header Files	**\INC**
On-line Help Files and Release Notes	**\HLP**
Common/Generic Example Programs	**\EXAMPLES**
Example Programs for Evaluation Boards	**\BOARDS**

Requesting Assistance

At Keil, we are committed to providing you with the best embedded development tools, documentation, and support. If you have suggestions and comments regarding any of our products, or you have discovered a problem with the software, please report them to us, and where applicable make sure to:

1. Read the section in this manual that pertains to the task you are attempting
2. Check the update section of the Keil web site to make sure you have the latest software and utility version
3. Isolate software problems by reducing your code to as few lines as possible

If you are still having difficulties, please report them to our technical support group. Make sure to include your license code and product version number. See the **Help – About** Menu. In addition, we offer the following support and information channels, all accessible at **www.keil.com/support**[1]**.**

1. The **Support Knowledgebase** is updated daily and includes the latest questions and answers from the support department
2. The **Application Notes** can help you in mastering complex issues, like interrupts and memory utilization
3. Check the on-line **Discussion Forum**
4. Request assistance through **Contact Technical Support** (web-based E-Mail)
5. Finally, you can reach the support department directly via **support.intl@keil.com** or **support.us@keil.com**

[1] *You can always get technical support, product updates, application notes, and sample programs at* ***www.keil.com/support.***

Chapter 2. Microcontroller Architectures

The Keil µVision Integrated Development Environment (µVision IDE) supports three major microcontroller architectures and sustains the development of a wide range of applications.

- **8-bit (classic and extended 8051)** devices include an efficient interrupt system designed for real-time performance and are found in more than 65% of all 8-bit applications. Over 1000 variants are available, with peripherals that include analog I/O, timer/counters, PWM, serial interfaces like UART, I^2C, LIN, SPI, USB, CAN, and on-chip RF transmitter supporting low-power wireless applications. Some architecture extensions provide up to 16MB memory with an enriched 16/32-bit instruction set.

  The µVision IDE supports the latest trends, like custom chip designs based on IP cores, which integrate application-specific peripherals on a single chip.

- **16-bit (Infineon C166, XE166, XC2000)** devices are tuned for optimum real-time and interrupt performance and provide a rich set of on-chip peripherals closely coupled with the microcontroller core. They include a Peripheral Event Controller (similar to memory-to-memory DMA) for high-speed data collection with little or no microcontroller overhead.

  These devices are the best choice for applications requiring extremely fast responses to external events.

- **32-bit (ARM7 and ARM9 based)** devices support complex applications, which require greater processing power. These cores provide high-speed 32-bit arithmetic within a 4GB address space. The RISC instruction set has been extended with a Thumb mode for high code density.

  ARM7 and ARM9 devices provide separate stack spaces for high-speed context switching enabling efficient multi-tasking operating systems. Bit-addressing and dedicated peripheral address spaces are not supported. Only two interrupt priority levels, - Interrupt Request (IRQ) and Fast Interrupt Request (FIQ), are available.

- **32-bit (Cortex-Mx based)** devices combine the cost benefits of 8-bit and 16-bit devices with the flexibility and performance of 32-bit devices at extremely low power consumption. The architecture delivers state of the art implementations for FPGAs and SoCs. With the improved Thumb2 instruction set, Cortex-Mx[1] based microcontrollers support a 4GB address space, provide bit-addressing (bit-banding), and several interrupts with at least 8 interrupt priority levels.

Selecting an Architecture

Choosing the optimal device for an embedded application is a complex task. The Keil Device Database (**www.keil.com/dd**) supports you in selecting the appropriate architecture and provides three different methods for searching. You can find your device by architecture, by specifying certain characteristics of the microcontroller, or by vendor.

The following sections explain the advantages of the different architectures and provide guidelines for finding the microcontroller that best fits your embedded application.

8051 Architecture Advantages

- Fast I/O operations and fast access to on-chip RAM in data space
- Efficient and flexible interrupt system
- Low-power operation

8051-based devices are typically used in small and medium sized applications that require high I/O throughput. Many devices with flexible peripherals are available, even in the smallest chip packages.

[1] *Cortex-M0 devices implement the Thumb instruction set.*

8051 Memory Types

A memory type prefix is used to assign a memory type to an expression with a constant. This is necessary, for example, when an expression is used as an address for the output command. Normally, symbolic names have an assigned memory type, so that the specification of the memory type can be omitted. The following memory types are defined:

Prefix	Memory Space
C:	Code Memory (CODE)
D:	Internal, direct-addressable RAM memory (DATA)
I:	Internal, indirect-addressable RAM memory (IDATA)
X:	External RAM memory (XDATA)
B:	Bit-addressable RAM memory
P:	Peripheral memory (VTREGD – 80x51 pins)

The prefix **P:** is a special case, since it always must be followed by a name. The name in turn is searched for in a special symbol table that contains the register's pin names.

Example:

C:0x100	Address 0x100 in CODE memory
ACC	Address 0xE0 in DATA memory, D:
I:100	Address 0x64 in internal RAM
X:0FFFFH	Address 0xFFFF in external data memory
B:0x7F	Bit address 127 or 2FH.7
C	Address 0xD7 (PSW.7), memory type B:

Infineon C166, XE166, XC2000

The 16-bit architecture of these devices is designed for high-speed real-time applications. It provides up to 16MB memory space with fast memory areas mapped into parts of the address space. High-performance applications benefit from locating frequently used variables into the fast memory areas. The below listed memory types address the following memory regions:

Memory Type	Description
bdata	Bit-addressable part of the **idata** memory.
huge	Complete 16MB memory with fast 16-bit address calculation. Object size limited to 64KB.
idata	High speed RAM providing maximum access speed (part of **sdata**).
near	Efficient variable and constant addressing (max. 64KB) with 16-bit pointer and 16-bit address calculation.
sdata	System area includes Peripheral Registers and additional on-chip RAM space.
xhuge	Complete 16MB memory with full address calculation for unlimited object size.

C166, XE166, XC2000 Highlights

- Highest-speed interrupt handling with 16 priority levels and up to 128 vectored interrupts
- Unlimited register banks for minimum interrupt prolog/epilog
- Bit instructions and bit-addressable space for efficient logical operations
- ATOMIC instruction sequences are protected from interrupts without interrupt enable/disable sequences
- Peripheral Event Controller (PEC) for automatic memory transfers triggered by peripheral interrupts. Requires no processor interaction and further improves interrupt response time.
- Multiply-Accumulate Unit (MAC) provided for high-speed DSP algorithms

C166, XE166, XC2000 Development Tool Support

The Keil C166 Compiler supports all C166, XE166, XC2000 specific features and provides additional extensions such as:

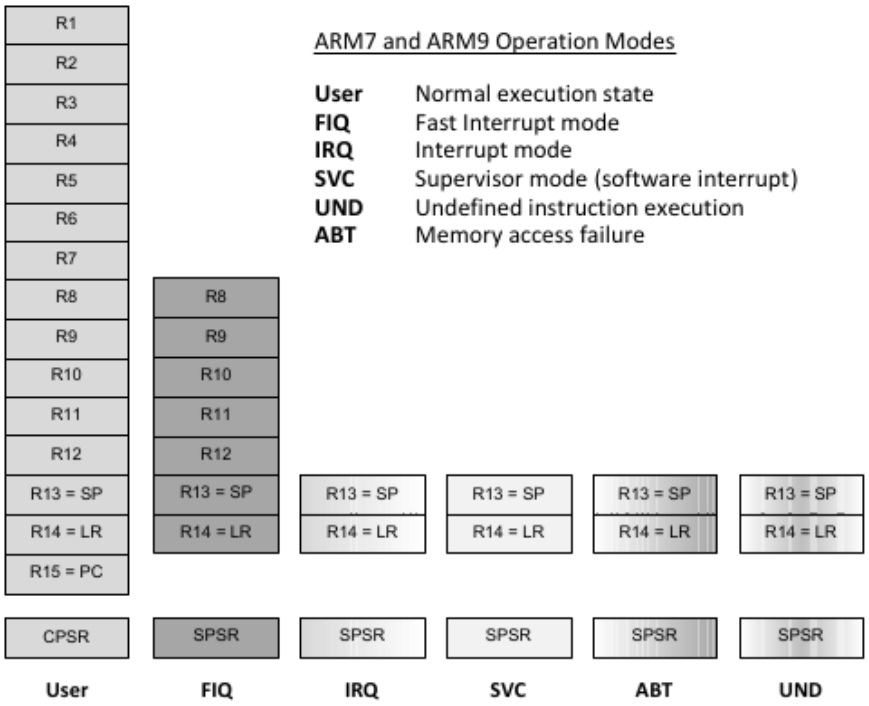
- Memory type support and flexible digital pattern processing for extremely fast variable access
- Function inlining eliminating call/return overhead
- Inline assembly for accessing all microcontroller and MAC instructions

ARM7 and ARM9 based Microcontrollers

The ARM7 and ARM9 based microcontrollers run on a load-store RISC architecture with 32-bit registers and fixed op-code length. The architecture provides a linear 4GB memory address space. In contrast to the previously mentioned 8/16-bit devices, no specific memory types are provided, since memory addressing is performed via 32-bit pointers in microcontroller registers. Peripheral registers are mapped directly into the linear address space. The Thumb instruction set improves code density by providing a compressed 16-bit instruction subset.

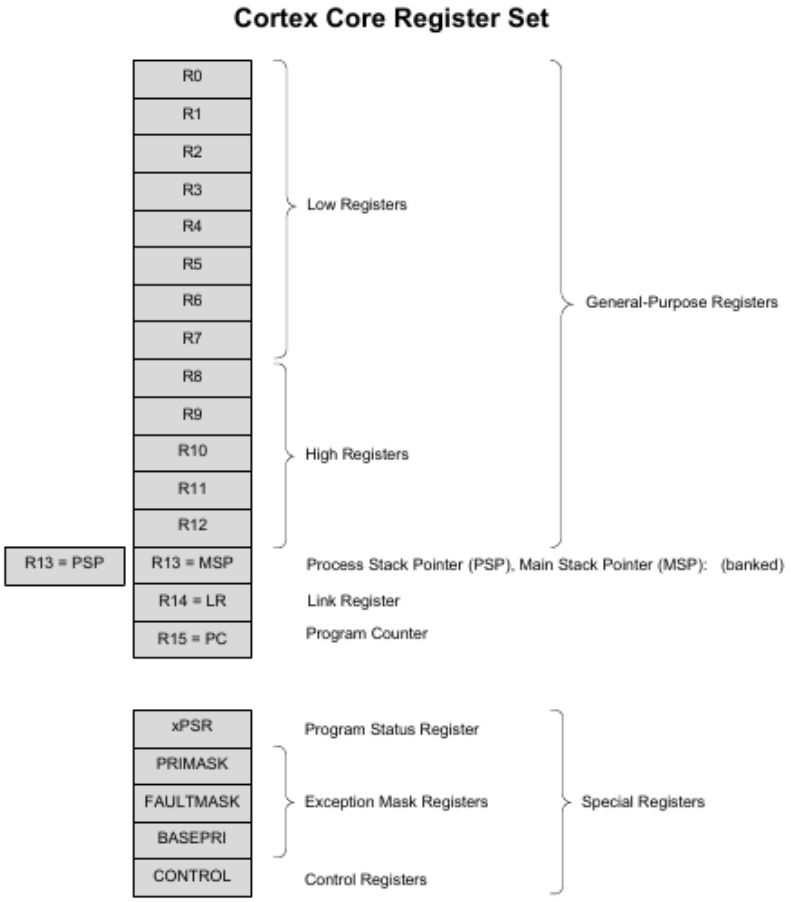
The ARM7 and ARM9 cores are easy to use, cost-effective, and support modern object-oriented programming techniques. They include a 2-level interrupt system with a normal interrupt (IRQ) and a fast interrupt (FIQ) vector. To minimize interrupt overhead, typical ARM7/ARM9 microcontrollers provide a vectored interrupt controller. The microcontroller operating modes, separate stack spaces, and Software Interrupt (SVC) features produce efficient use of Real-Time Operating Systems.

The ARM7 and ARM9 core provides thirteen general-purpose registers (R0–R12), the stack pointer (SP) R13, the link register (LR) R14, which holds return addresses on function calls, the program counter (PC) R15, and a program status register (PSR). Shadow registers, available in various operating modes, are similar to register banks and reduce interrupt latency.



ARM7 and ARM9 Highlights

- **Linear 4 GB memory space** that includes peripherals and eliminates the need for specific memory types
- **Load-store architecture with efficient pointer addressing**. Fast task context switch times are achieved with multiple register load/store.
- **Standard (IRQ) and Fast (FIQ) interrupt**. Banked microcontroller registers on FIQ reduce register save/restore overhead.
- **Vectored Interrupt Controller** (available in most microcontrollers) optimizes multiple interrupt handling
- **Processor modes** with separate interrupt stacks for predictable stack requirements
- **Compact 16-bit Instruction Set (Thumb)**. Compared to ARM mode, Thumb mode code is about 65% of the code size and 160% faster when executing from a 16-bit memory system.



Cortex-Mx Highlights

- **Nested Vectored Interrupt Controller** optimizes multiple external interrupts (up to 240 + 1 NMI, with at least eight priority levels)
- **R0-R3**, **R12**, **LR**, **PSR**, **and PC are pushed automatically** to the stack at interrupt entry and popped back at interrupt exit points
- **Only one instruction set (Thumb2)**, assuring software upward compatibility with the entire ARM roadmap
- **Several Extreme Low-Power Modes** with an attached Wake-Up Interrupt Controller (WIC)

Cortex-Mx Development Tool Support

In addition to the ARM specific characteristics, the Keil MDK-ARM supports the Cortex-Mx Microcontroller Software Interface Standard (CMSIS) and provides the following features:

- **Core registers and core peripherals** are accessible through C/C++ functions
- Device independent debug channel for RTOS kernels
- **Supports object oriented programming**, reuse of code, and implements an easy way of porting code to different devices
- **Extensive debug capabilities** allowing direct access to memory without stopping the processor
- **CMSIS is supported**, making the software compatible across the Cortex-Mx architectures

Architecture Comparison Conclusions

The various architectures have pros and cons and the optimal choice depends highly on the application requirements. The following code comparison section provides additional architectural information that can help you in selecting the optimal microcontroller for your target embedded system.

Code Comparison

The following short but representative code examples show the impressive individual strengths of the different microcontroller architectures.

I/O Port Access Comparison

Source Code	Description
`if (IO_PIN == 1) { i++; }`	Increment a value when an I/O pin is set.

- **8051** devices provide bit-addressable I/O Ports and instructions to access fixed memory locations directly
- **C166**, **XE166**, **XC2000** devices provide bit-addressable I/O Ports and instructions to access fixed memory locations directly
- **ARM7 and ARM9** devices provide indirect memory access instructions only. However, there are no bit operations.
- **Cortex-Mx** devices provide indirect memory access instructions only, but allow atomic bit operations

8051 Code	C166/XE166 and XC2000 Code	ARM7 and ARM9 Thumb Code	Cortex-Mx Thumb2 Code

`sfr P0=0x80;`
`sbit P0_0=P0^0;`
 	
`unsigned char i;`
 	
`void main (void) {`
`if (P0_0) {`
`; JNB P0_0,?C0002`
 	
`i++;`
`; INC i`
 	
`}`
 	
`; RET`
`}`
	`sfr P0L=0xFF00;`
	`sbit P0_0=P0L^0;`
	
	`unsigned int i;`
	
	`void main (void) {`
	`if (P0_0) {`
	`; JNB P0_0,?C0001`
	
	`i++;`
	`; SUB i,ONES`
	
	`}`
	
	`; RET`
	`}`
		`#define IOP *(int*))`
		
		`unsigned int i;`
		
		`void main (void) {`
		`if (IOP & 1) {`
		`; LDR R0,=0xE0028000`
		`; LDR R0,[R0,#0x0]`
		`; MOV R1,#0x1`
		`; TST R0,R1`
		`; BEQ L_1`
		
		`i++;`
		`; LDR R0,=i ; i`
		`; LDR R1,[R0,#0x0];i`
		`; ADD R1,#0x1`
		`; STR R1,[R0,#0x0];i`
		`}`
		
		`; BX LR`
		`}`
			
			`unsigned int i;`
			
			`void main (void) {`
			`if (GPIOA->ODR) {`
			`; STR R0,[R1,#0xc]`
			`; LDR R0,[R2,#0]`
			`; CBZ R0,|L1.242|`
			
			`i++;`
			`; MOVS R0,#2`
			`; STR R0,[R1,#0xc]`
			`; |L1.242|`
			
			`}`
			
			`; BX LR`
			`}`

6 Bytes	10 Bytes	24 Bytes	12 Bytes

Pointer Access Comparison

Source Code	Description

`typedef struct { int x; int arr[10]; } sx;`
 	
`int f (sx xdata *sp, int i) {`
`  return sp->arr[i];`
`}`
	Return a value that is part of a struct and indirectly accessed via pointer.

- **8051** devices provide byte arithmetic requiring several microcontroller instructions for address calculation
- **C166**, **XE166**, **XC2000** devices provide efficient address arithmetic with direct support of a large 16 MByte address space
- **ARM** devices are extremely efficient with regard to pointer addressing and always use the 32-bit addressing mode
- In **Cortex-Mx** devices, any register can be used as a pointer to data structures and arrays

8051 Code	C166, XE166, XC2000 Code	ARM 7 and ARM9 Thumb Code	Cortex-Mx Thumb2 Code

`MOV DPL,R7`
`MOV DPH,R6`
`MOV A,R5`
`ADD A,ACC`
`MOV R7,A`
`MOV A,R4`
`RLC A`
`MOV R6,A`
`INC DPTR`
`INC DPTR`
`MOV A,DPL`
`ADD A,R7`
`MOV DPL,A`
`MOV A,DPH`
`ADDC A,R6`
`MOV DPH,A`
`MOVX A,@DPTR`
`MOV R6,A`
`INC DPTR`
`MOVX A,@DPTR`
`MOV R7,A`
	`MOV R4,R10`
	`SHL R4,#01H`
	`ADD R4,R8`
	`EXTS R9,#01H`
	`MOV R4,[R4+#2]`
		`LSL R0,R1,#0x2`
		`ADD R0,R2,R0`
		`LDR R0,[R0,#0x4]`
			`ADD R0,R0,R1,LSL #2`
			`LDR R0,[R0,#4]`

25 Bytes	14 Bytes	6 Bytes	6-Bytes

The µVision Debugger offers two operating modes—**Simulator Mode** and **Target Mode**.

Simulator Mode configures the µVision Debugger as a *software-only product* that accurately simulates target systems including instructions and most on-chip peripherals. In this mode, you can test your application code before any hardware is available. It gives you serious benefits for rapid development of reliable embedded software. The Simulator Mode offers:

- Software testing on your desktop with no hardware environment
- Early software debugging on a functional basis improves software reliability
- Breakpoints that are impossible with hardware debuggers
- Optimal input signals. Hardware debuggers add extra noise
- Single-stepping through signal processing algorithms is possible. External signals are stopped when the microcontroller halts.
- Detection of failure scenarios that would destroy real hardware peripherals

Target Mode[1] connects the µVision Debugger to *real hardware*. Several target drivers are available that interface to a:

- **ULINK JTAG/OCDS Adapter** that connects to on-chip debugging systems
- **Monitor** that may be integrated with user hardware or that is available on many evaluation boards
- **Emulator** that connects to the microcontroller pins of the target hardware
- **In-System Debugger** that is part of the user application program and provides basic test functions
- **ULINK*Pro* Adapter** a high-speed debug and trace unit connecting to on-chip debugging systems via JTAG/SWD/SWV, and offering Cortex-M3 ETM Instruction Trace capabilities

[1] *Some target drivers have hardware restrictions that limit or eliminate features of the µVision Debugger while debugging the target hardware.*

Assembler

An assembler allows you to write programs using microcontroller instructions. It is used where utmost speed, small code size, and exact hardware control is essential. The Keil Assemblers translate symbolic assembler language mnemonics into executable machine code while supporting source-level symbolic debugging. In addition, they offer powerful capabilities like macro processing.

The assembler translates assembly source files into re-locatable object modules and can optionally create listing files with symbol table and cross-reference details. Complete line number, symbol, and type information is written to the generated object files. This information enables the debugger to display the program variables exactly. Line numbers are used for source-level debugging with the µVision Debugger or other third-party debugging tools.

Keil assemblers support several different types of macro processors (depending on architecture):

- The **Standard Macro Processor** is the easier macro processor to use. It allows you to define and use macros in your assembly programs using syntax that is compatible with that used in many other assemblers.
- The **Macro Processing Language or MPL** is a string replacement facility that is compatible with the Intel ASM-51 macro processor. MPL has several predefined macro processor functions that perform useful operations like string manipulation and number processing.

Macros save development and maintenance time, since commonly used sequences need to be developed once only.

Another powerful feature of the assembler's macro processor is the conditional assembly capability. You can invoke conditional assembly through command line directives or symbols in your assembly program. Conditional assembly of code sections can help achieve the most compact code possible. It also allows you to generate different applications from a single assembly source file.

C/C++ Compiler

The ARM C/C++ compiler is designed to generate fast and compact code for the ARM7, ARM9 and Cortex-Mx processor architectures; while the Keil ANSI C compilers target the 8051, C166, XE166, and XC2000 architectures. They can generate object code that matches the efficiency and speed of assembly programming. Using a high-level language like C/C++ offers many advantages over assembly language programming:

- Knowledge of the processor instruction set is not required. Rudimentary knowledge of the microcontroller architecture is desirable, but not necessary.
- Details, like register allocation, addressing of the various memory types, and addressing data types, are managed by the compiler
- Programs receive a formal structure (imposed by the C/C++ programming language) and can be split into distinct functions. This contributes to source code reusability as well as a better application structure.
- Keywords and operational functions that resemble the human thought process may be used
- Software development time and debugging time are significantly reduced
- You can use the standard routines from the run-time library such as formatted output, numeric conversions, and floating-point arithmetic
- Through modular programming techniques, existing program components can be integrated easily into new programs
- The C/C++ language is portable (based on the ANSI standard), enjoys wide and popular support, and is easily obtained for most systems. Existing program code can be adapted quickly and as needed to other processors.

Object-HEX Converter

The object-hex converter creates Intel HEX files from absolute object modules that have been created by the linker. Intel HEX files are ASCII files containing a hexadecimal representation of your application program. They are loaded easily into a device program for writing to ROM, EPROM, FLASH, or other programmable memory. Intel HEX files can be manipulated easily to include checksum or CRC data.

Linker/Locator

The linker/locator combines object modules into a single, executable program. It resolves external and public references and assigns absolute addresses to relocatable program segments. The linker includes the appropriate run-time library modules automatically and processes the object modules created by the Compiler and Assembler. You can invoke the linker from the command line or from within the µVision IDE. To accommodate most applications, the default linker directives have been chosen carefully and need no additional options. However, it is easy to specify additional custom settings for any application.

Library Manager

The library manager creates and maintains libraries of object modules (created by the C/C++ Compiler and Assembler). Library files provide a convenient way to combine and reference a large number of modules that may be used by the linker.

The linker includes libraries to resolve external variables and functions used in applications. Modules from libraries are extracted and added to programs only if required. Modules, containing routines that are not invoked by your program specifically, are not included in the final output. Object modules extracted by the linker from a library are processed exactly like other object modules.

There are a number of advantages to using libraries: security, speed, and minimized disk space are only a few. Libraries provide a vehicle for distributing large numbers of functions and routines without distributing the original source code. For example, the ANSI C library is supplied as a set of library files.

You can build library files (instead of executable programs) using the µVision **Project Manager**. To do so, check the **Create Library** check box in the **Options for Target — Output** dialog. Alternatively, you may invoke the library manager from the **Command Window**.


Navigate Forwards – moves cursor to its former forward position


Bookmark – sets or removes a bookmark at cursor position


Previous Bookmark – moves the cursor to the bookmark previous to the current cursor position


Next Bookmark – moves cursor to the bookmark ahead of the current cursor position


Clear All Bookmarks – removes bookmarks in the current document


Indent – moves the lines of the highlighted text one tab stop to the right


Unindent – moves all highlighted text lines one tab stop to the left


Set Comment – converts the selected code/text to comment lines


Remove Comment – converts the selected text lines back to code lines


Find in Files – searches for text in files; results shown in an extra window


Find – searches for specified text in current document


Incremental Find – finds expression as you type


Debug Session – starts/stops debugging


Breakpoint – sets or removes a breakpoint at cursor position


Disable Breakpoint – disables the breakpoint at cursor position


Disable All Breakpoints – disables all breakpoints in all documents


Kill All Breakpoints – removes all breakpoints from all documents


Project Window – dropdown to enable/disable project related windows


Configure – dialog to configure your editor, shortcuts, keywords, …

Build Toolbar

 Translate/Compile – compiles or assembles the file in the current edit window

 Build – builds and links those files of the project that have changed or whose dependencies have changed

 Rebuild – re-compiles, re-assembles, and re-links all files of the project

 Batch Build – re-builds the application based on batch instructions. This feature is active in a Multi-Project environment only.

 Stop Build – halts the build process

 Download – downloads your application to the target system flash

 Target – drop-down box to select your target system (in the Toolbar example above: Simulator)

 Target Options – dialog to define tool and target settings. Set device, target, compiler, linker, assembler, and debug options here. You can also configure your flash device from here.

 File Extensions, Environments, and Books – dialog to configure targets, groups, default folders, file extensions, and additional books

 Manage Multi-Project Workspace – dialog to add or remove individual projects or programs to or from your multi-project container

Debug Toolbar


Reset – Resets the microcontroller CPU or simulator while debugging


Run – continues target program execution to next breakpoint


Stop – halts target program execution


Step One Line – steps to the next instruction or into procedure calls


Step Over – steps over a single instruction and over procedure calls


Step Out – steps out of the current procedure


Run to Line – runs the program until the current cursor line


Show Current Statement – Shows next statement to be executed


Command Window – displays/hides the Command Window


Disassembly Window – displays/hides the Disassembly Window


Symbol Window – displays/hides Symbols, Variables, Ports, …


Register Window – displays/hides Registers


Call Stack Window – displays/hides the Call Stack tree


Watch Window – drop-down to display/hide Locals and Watch Windows


Memory Window – drop-down to display/hide Memory Windows


Serial Window – drop-down to display/hide UART-peripheral windows and the Debug printf() View


Logic Analyzer – displays variable values graphically; Also used as a drop-down to display/hide the Performance Analyzer and Code Coverage Window.

 Performance Analyzer – displays, in graphical form, the time consumed by modules and functions as well as the number of function calls

 Code Coverage – dialog to view code execution statistics in a different way than with the Performance Analyzer

 System Viewer – view the values of your Peripheral Registers

 Instruction Trace – displays/hides the Instruction Trace Window

 Toolbox – shows/hides the Toolbox dialog. Depending on your target system, various options are available.

 Debug Restore Views – drop-down to select the preferred window layout while debugging

Additional Icons

 Print – opens the printer dialog

 Books – opens the Books Window in the Project Workspace

 Functions – opens the Functions Window in the Project Workspace

 Templates – opens the Templates Window in the Project Workspace

 Source Browser – opens the Source Browser Window in the Output Workspace. Use this feature to find definitions or occurrences of variables, functions, modules, and macros in your code.

 µVision Help – opens the µVision Help Browser

 File – Source file; you can modify these files; default options are used

 File – Source file; you can modify these files; file options have been changed and are different from the default options

 File or Module – header files; normally, included automatically into the project; options cannot be set for these file types

 Folder or Group – expanded – icon identifying an expanded folder or group; options correspond to the default settings

 Folder or group – expanded – icon identifying an expanded folder or group; with changed options that are different from the default settings

 Folder or group – collapsed – with options corresponding to default settings

 Folder or group – collapsed – with options changed and different form default settings

 Lock – freezes the content of a window; prevents that window from refreshing periodically; You cannot manually change the content of that window.

 Unlock – unfreezes the content of a window; allows that window to refresh periodically. You can manually change the content of that window.

 Insert – creates or adds an item or object to a list

 Delete - removes an item or object from a list

 Move Up - moves an item or object higher up in the list

 Move Down - moves an item or object down in the list

 Peripheral SFR (Peripheral Registers, Special Function Register)

 Simulator VTREG (Virtual Register)

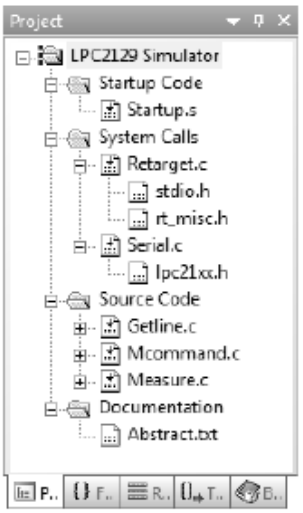
 Application, Container

 Variable

 Parameter

 Function

Project Windows

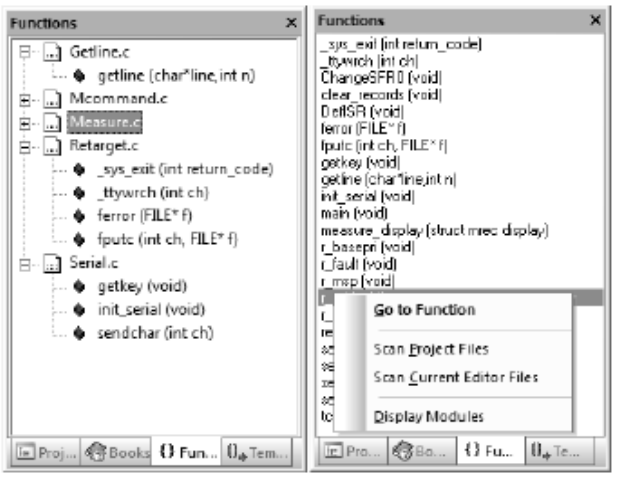


Project Windows display information about the current project. The tabs at the bottom of this area provide access to:

- **Project** structure and management. Group your files to enhance the project overview.
- **Functions** of the project. Quickly find and navigate between functions of the source code.
- Microcontroller **Registers**. Only available while debugging.
- **Templates** for often-used text blocks. Double click the definitions to insert the predefined text at cursor position.
- **Books** specific to the µVision IDE, the project, and sometimes to the microcontroller used. Configure and add your own books to any section.

The **Functions Window** displays all functions of your project or of open editor files.

Double-click a function to jump to its definition. Invoke its **Context Menu** to toggle the displaying mode of this window or scan the files.



The **Templates Window** provides user-defined text blocks, which can be defined through the **Configuration – Templates** dialog.

Double-click a definition or invoke the **Context Menu** to insert often-needed constructs into your code files.

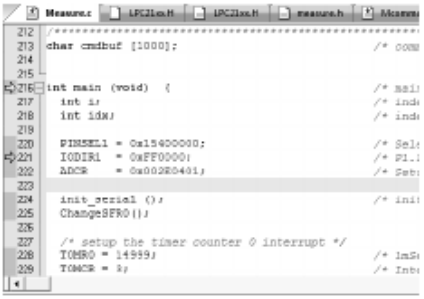
Alternatively, you can type the first few letters of the template name followed by **Ctrl+Space** to insert the text.

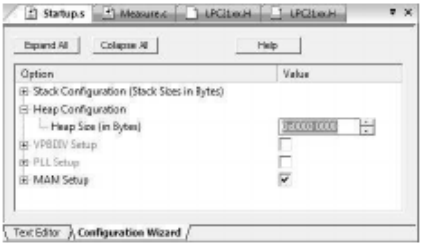
Editor Windows

The **Editor Windows** are used to:

- Write, edit, and debug source files. Press **F1** on language elements for help.
- Set breakpoints and bookmarks
- Set project options and initialize target systems by using powerful configuration wizards
- View disassembly code and trace instructions while debugging

Typically, this area contains the **Text Editor** with source code files, the **Disassembly Window**, **Performance Analyzer**, and **Logical Analyzer**.

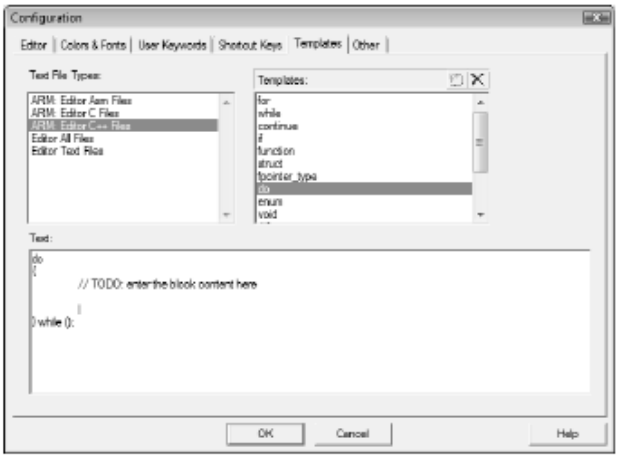




Editor Configuration

Configure Editor settings, colors and fonts, user defined keywords, shortcut keys, and templates through the **Configuration** dialog.

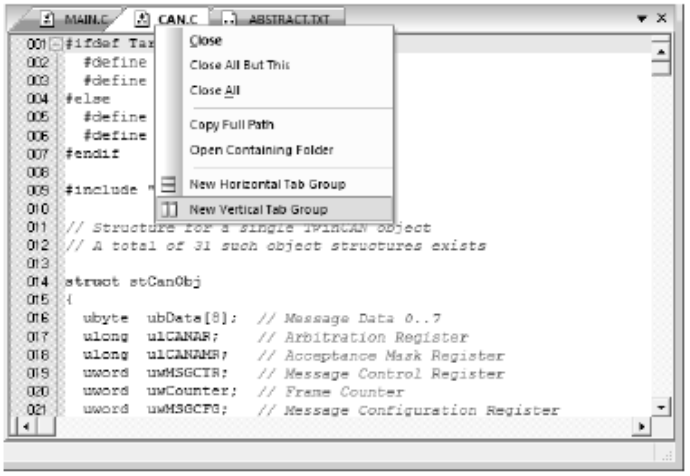
You can invoke this dialog via the **Context Menu** of the **Template Window**, the **Edit – Configuration** Menu, or



through the **File Toolbar** command.

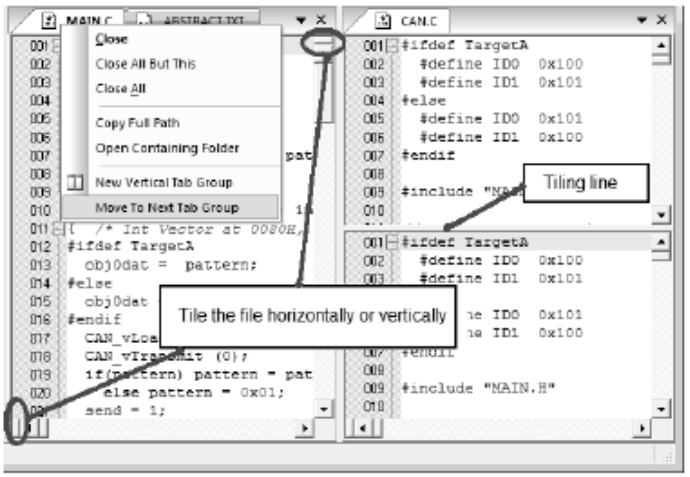
Using the Editor

You can view text files in the Editor side by side. Invoke the **Context Menu** of the file tab and choose a horizontal or vertical arrangement.



Files can be dragged and dropped from one **Tab Group** into the other, or can be moved to the Next Tab Group through the **Context Menu**.

In addition, you can tile a file vertically and horizontally. Complete your code in any part of these fragments.



Double-click the tiling line to remove the fragmentation.

Double-click a file's tab to close the file.

Output Windows

By default, the **Output Windows**[1] are displayed at the bottom of the µVision screen and include:

- The **Build Output Window** includes errors and warnings from the compiler, assembler, and linker. Double-click a message to jump to the location of the source code that triggered the message. Press **F1** for on-line help.
- The **Command Window** allows you to enter commands and review debugger responses. Hints are provided on the **Status Bar** of that window. Press **F1** for on-line help.
- The **Find in Files Window** allows you to double-click a result to locate the source code that triggered the message
- The **Serial** and **UART** windows display I/O information of your peripherals
- The **Call Stack Window** enables you to follow the program call tree
- The **Locals Window** displays information about local variables of the current function
- The **Watch** windows provide a convenient way to personalize a set of variables you would like to trace. Objects, structures, unions, and arrays may be monitored in detail.
- The **Symbols Window** is a handy option to locate object definitions. You can drag and drop these items into other areas of µVision.
- The **Memory** windows enable you to examine values in memory areas. Define your preferred address to view data.
- The **Source Browser Window** offers a fast way to find occurrences and definitions of objects. Enter your search criteria to narrow the output.

[1] *Since almost all objects can be moved to their own window frame, the terminology 'page' and 'window' is interchangeably used in this book.*

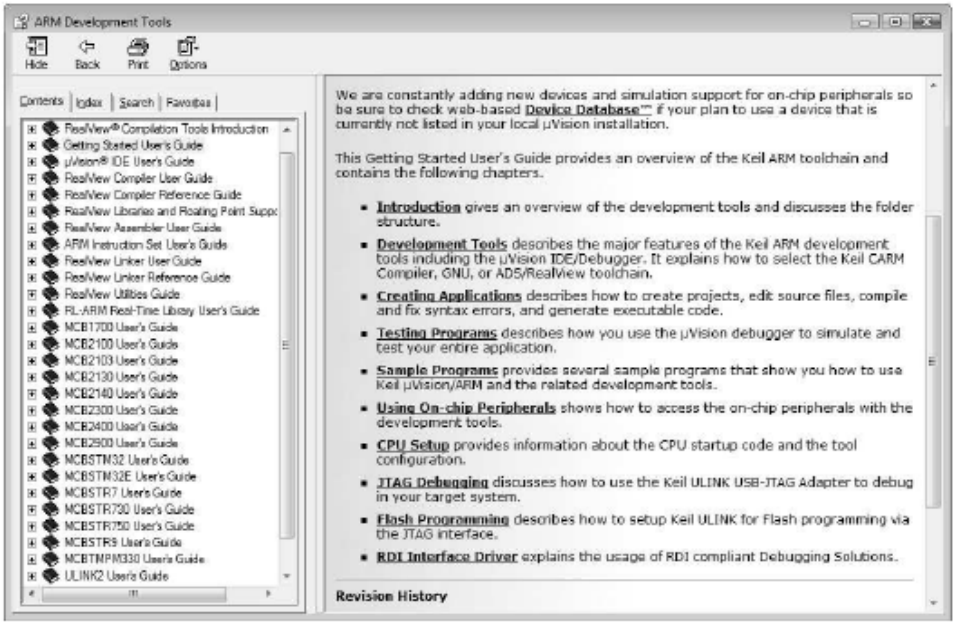
Other Windows and Dialogs

Peripheral Dialogs and Windows

Peripheral Dialogs and Windows allow you to review and modify the status of on-chip peripherals. These dialogs are dependent on the target system you selected at the beginning of your project and thus the options provided will vary.

On-line Help

µVision includes many pages of on-line manuals and context-sensitive help. The main help system is available from the **Help** Menu.



Context sensitive on-line help is available in most dialogs in µVision. Additionally, you can press **F1** in the **Editor Windows** for help on language elements like compiler directives and library routines. Use **F1** in the **Output Window** for help on debug commands, error messages, and warnings.

Chapter 6. Creating Embedded Programs

µVision is a Windows application that encapsulates the Keil microcontroller development tools as well as several third-party utilities. µVision provides everything you need to start creating embedded programs quickly.

µVision includes an advanced editor, project manager, and make utility, which work together to ease your development efforts, decreases the learning curve, and helps you to get started with creating embedded applications quickly.

There are several tasks involved in creating a new embedded project:

- Creating a Project File
- Using the Project Windows
- Creating Source Files
- Adding Source Files to the Project
- Using Targets, Groups, and Files
- Setting Target Options, Groups Options, and File Options
- Configuring the Startup Code
- Building the Project
- Creating a HEX File
- Working with Multi-Projects

The section provides a step-by-step tutorial that shows you how to create an embedded project using the µVision IDE.

Creating a Project File

Creating a new µVision project requires just three steps:

1. Select the Project Folder and Project Filename
2. Select the Target Microcontroller
3. Copy the Startup Code to the Project Folder

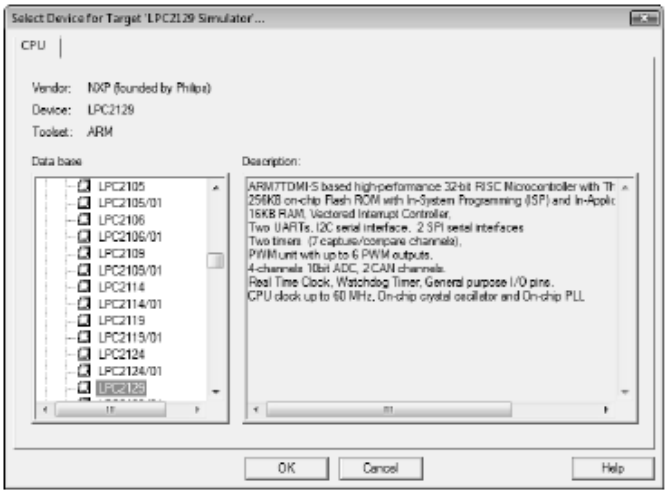
Selecting the Folder and Project Name

To create a new project file, select the **Project – New Project…** Menu. This opens a standard dialog that prompts you for the new project file name. It is good practice to use a separate folder for each project. You may use the **Create New Folder** button in this dialog to create a new empty folder.

Select the preferred folder and enter the file name for the new project. µVision creates a new, empty project file with the specified name. The project contains a default target and file group name, which you can view on the **Project Window**.

Selecting the Target Microcontroller

After you have selected the folder and decided upon a file name for the project, µVision asks you to choose a target microcontroller. This step is very important, since µVision customizes the tool settings, peripherals, and dialogs for that particular device.



The **Select Device**[1,2] dialog box lists all the devices from the µVision **Device Database.**

You may invoke this screen through the **Project – Select Device for Target…** Menu in order to change target later.

---

[1] *For some devices, µVision requires additional parameters you must enter manually. Please read the device description in the Select Device dialog carefully, as it may contain extra instructions for the device configuration.*

[2] *If you do not know the actual device you will finally use, µVision allows you to change the device settings for a target after project creation.*

Copying the Startup Code

All embedded programs require some kind of microcontroller initialization or startup code[1,2] that is dependent of the tool chain and hardware you will use. It is required to specify the starting configuration of your hardware.

All Keil tools include chip-specific startup code for most of the devices listed in the **Device Database**. Copy the startup code to your project folder and modify it there only. µVision automatically displays a dialog to copy the startup code into your project folder. Answer this question with **YES**. µVision will copy the startup code to your project folder and adds the startup file to the project.

The startup code files are delivered with embedded comments used by the configuration wizard to provide you with a GUI interface for startup configuration.

Using the Project Windows

Once you have created a project file successfully, the **Project Window** shows the targets, groups, and files of your project. By default, the target name is set to **Target 1**, while the group's name is **Source Group 1**.

The file containing the startup code is added to the source group. Any file, the startup file included, may be moved to any other group you may define in future.

The **Books Window**, also part of the **Project Windows**, provides the Keil product manuals, data sheets, and programmer's guides for the selected microcontroller. Double-click a book to open it.

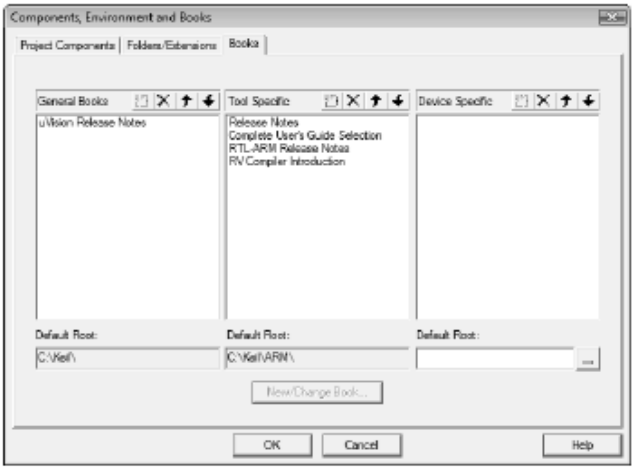
---

[1] *The startup code's default settings provide a good starting point for most single-chip applications. However, changes to the startup code may be required.*

[2] *Library and add-on projects need no startup code.*

Right-click the **Books Window** to open its **Context Menu**. Choose **Manage Books…**, to invoke the

Components, Environments and Books[1] dialog to modify the settings of the exiting manuals or add your own manuals to the list of books.



Later, while developing the program, you may use the **Functions Window** and **Templates Window** as well.

Creating Source Files

Use the button on the **File Toolbar** or the select the **File – New…** Menu to create a new source file

This action opens an empty **Editor Window** to enter your source code. µVision enables color syntax highlighting based on the file extension (after you have saved the file). To use this feature immediately, save the empty file with the desired extension prior to starting coding.

Save the new source file using the button on the **File Toolbar** or use the **File** – **Save** Menu

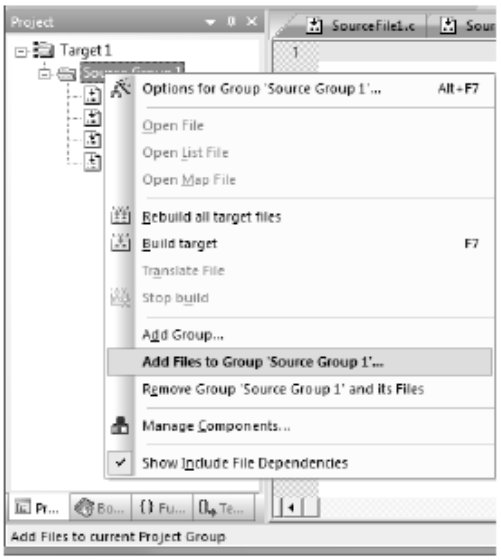
[1] *Most microcontroller manuals are part of the toolset, or are available on the Keil Development Tools CD-ROM.*

Adding Source Files to the Project

After you have created and saved your source file, add it to the project. Files existing in the project folder, but not included in the current project structure, will not be compiled.

Right-click a file group in the **Project Window** and select **Add Files to Group** from the **Context Menu**. Then, select the source file or source files to be added.

A self-explanatory window will guide you through the steps of adding a file.



Using Targets, Groups, and Files

The µVision's very flexible project management structure allows you to create more than one **Target** for the same project.

A **Target** is a defined set of build options that assemble, compile, and link the included files in a specific way for a specific platform.

Multiple file groups may be added to a target and multiple files may be attached to the same file group.

You can define **multiple targets** for the same project as well.

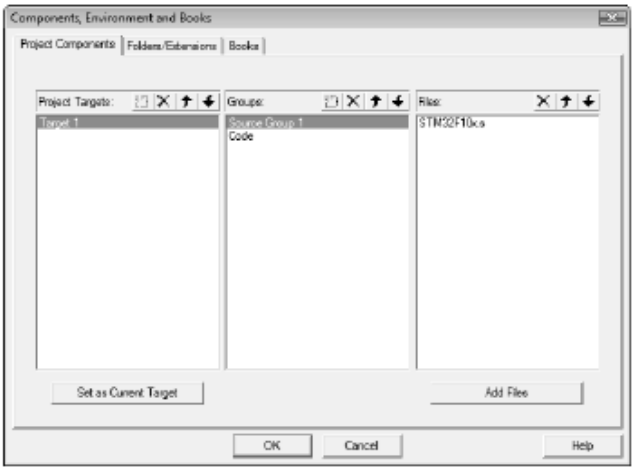
You should customize the name of targets and groups to match your application structure and internal naming conventions. It is a good practice to create a separate file group for microcontroller configuration files.

Use the **Components, Environment, and Books…** dialog to manage your Targets, Groups, and Files configuration

To change the name of a Target, Group, or File you may either:

- Double-click the desired item, or
- Highlight the item and press **F2**

Change the name and click the **OK** button. Changes will be visible in the other windows as soon as this dialog has been closed.



 Insert - create a new target or group

 Delete - remove a target, group, or source file from the project

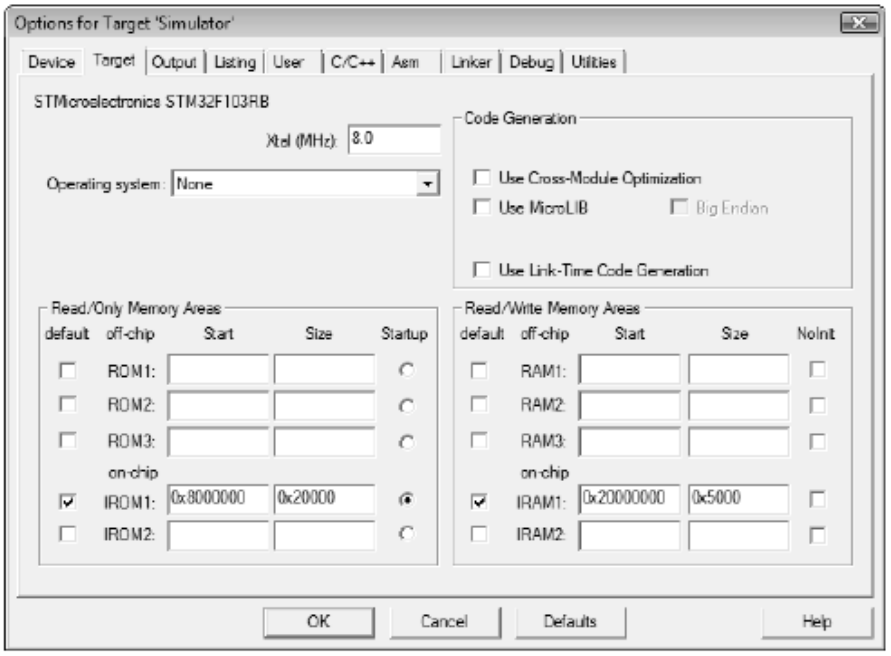
 Move Up - move a target, group, or source file up the list

 Move Down - move a target, group, or source file down the list

Instead of using the Move Up or Move Down buttons, you may drag and drop the source files within the **Project Window** to re-arrange the order of the files.

Setting Target Options

Open the **Options for Target** dialog from the **Build Toolbar** or from the **Project** Menu



Through this dialog, you can

- change the target device
- set target options
- and configure the development tools and utilities

Normally, you do not have to make changes to the default settings in the **Target** and **Output** dialog.

The options available in the **Options for Target** dialogs depend on the microcontroller device selected. Of course, the available tabs and pages will change in accordance with the device selected and with the target.

When switching between devices, the menu options are available as soon as the **OK** button in the **Device Selection** dialog has been clicked.

The following table lists the project options that are configurable on each page of the **Target Options** dialog.

Dialog Page	Description
Device	Selects the target device from the Device Database
Target	Specifies the hardware settings of your target system
Output	Specifies the output folders and output files generated
Listing	Specifies the listing folders and listing files generated
User	Allows you to start user programs before and after the build process
C/C++	Sets project-wide C/C++ Compiler options
Asm	Sets project-wide Assembler options
Linker	Sets project-wide Linker options. Linker options are typically required to configure the physical memory of the target system and locate memory classes and sections.
Debug	Sets Debugger options, including whether to use hardware or simulation
Utilities	Configures utilities for Flash programming

Setting Group and File Options

In µVision, properties of objects and options can be set at the group level and on individual files. Use this powerful feature to set options for files and groups that need a configuration different from the default settings. To do so, open the **Project Window**:

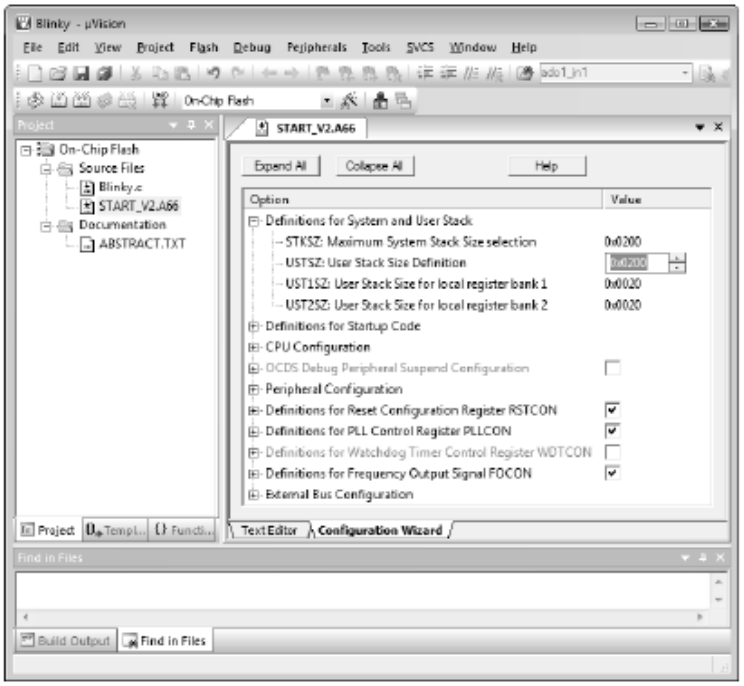
- Invoke the **Context Menu** of a file group and select **Options for Group** to specify the properties, compiler options, and assembler options for that file group
- Invoke the **Context Menu** of a source file and select **Options for File** to specify the properties, compiler, or assembler options for that file

Treat **Target** options similar to general options. They are valid for the entire project and for that target. Some options can be defined at the group level and on individual files. File-level options will supersede group-level options, which in turn, supersede the options set at the target level.

 Red dots on the icon's left side are an indication that the options of that item differ from the general target options

Configuring the Startup Code

Keil tools include files with chip-specific startup code for most of the supported devices.



Keil startup files contain assembler code with options you can adjust to your particular target system. Most startup files have embedded commands for the µVision **Configuration Wizard**, which provides an intuitive, graphical, and convenient interface to edit the startup code.

Simply click the desired value to change data. Alternatively, you can use the **Text Editor** to directly edit the assembly source file.

Keil startup files provide a good starting point for most single-chip applications. However, you must adapt their configuration for your target hardware. Target-specific settings, like the microcontroller PLL clock and BUS system, have to be configured manually.

Building the Project

Several commands are available from the **Build Toolbar** or **Project** Menu to assemble, compile, and link the files of your project. Before any of these actions are executed, files are saved.

Translate File – compiles or assembles the currently active source file

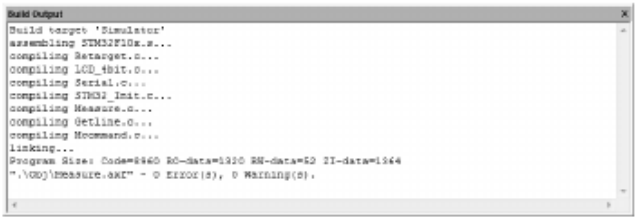
Build Target – compiles and assembles those files that have changed, then links the project

Rebuild – compiles and assembles all files, regardless whether they have changed or not, then links the project

While assembling, compiling, and linking, µVision displays errors and warnings in the **Build Output Window**.

Highlight an error or warning and press **F1** to get help regarding that particular message. Double-click the message to jump to the source line that caused the error or warning.

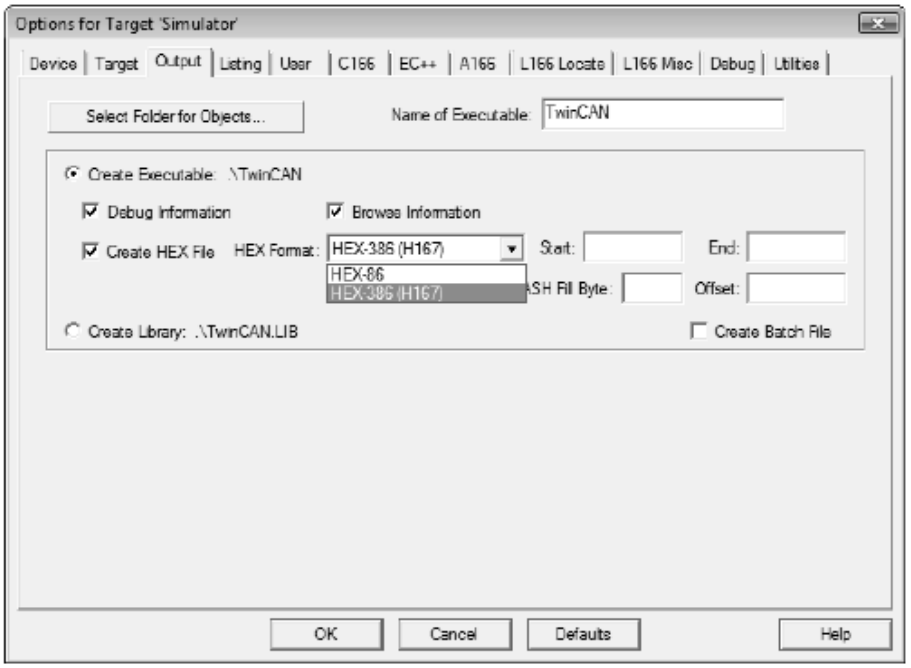
µVision displays the message **0 Error(s), 0 Warning(s)** on successful completion of the build process. Though existing warnings do not prevent the program from running correctly, you should consider solving them to eliminate unwanted effects, such as time consumption, undesirable side effects, or any other actions not necessary for your program.



Creating a HEX File

Check the **Create HEX File** box under **Options for Target — Output**, and µVision will automatically create a HEX file during the build process.

Select the desired HEX format through the drop-down control to generate formatted HEX files, which are required on some Flash programming utilities.



Working with Multiple Projects

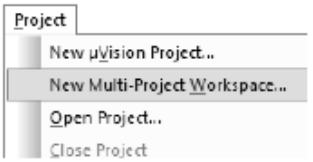
Sometimes, application development requires working with more than one project at the same time. With single projects, that requires closing the current project and opening the new project. The µVision Multi-Project feature allows you to define a group of projects as a Multi-Project file and to work with those projects in one Project Window.

By combining µVision projects, which logically depend on each other, into one **Multi-Project**, you increase the overview, consistency, and transparency of your embedded system application design. µVision supports you in grouping various stand-alone projects into one project overview.

While all features described for single-projects also apply to Multi-Projects, additional functionalities are required and are available in the µVision IDE.

Creating a Multiple Project

Choose **Project – New Multi-Project Workspace…** to create a new Multi-Project file. This opens a standard Windows dialog that prompts you for the new project file name.



To open an existing Multi-Project, choose **Project – Open Project**. You can differentiate a Multi-Project file from a stand-alone project file by its file extension. A file containing a Multi-Project has the extension *filename*.**uvmpv** rather than *filename*.**uvproj** – the naming convention for stand-alone projects.

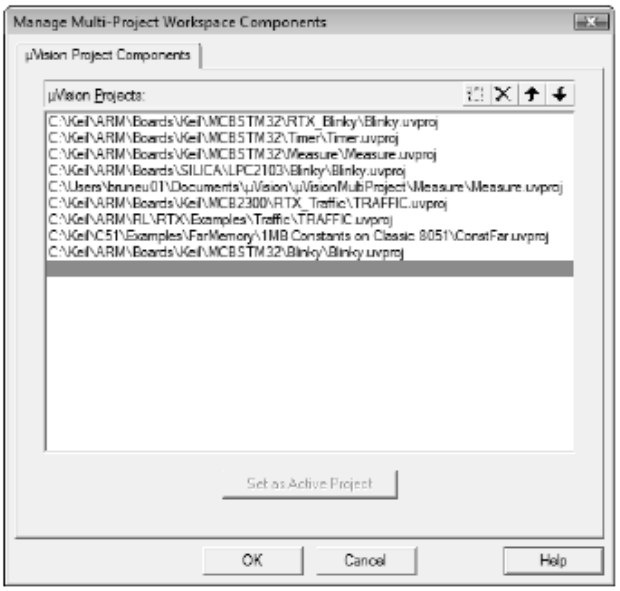
Managing Multiple Projects

Invoke the **Manage Multi-Project Workspace Components** dialog through the **Project – Manage – Multi Project Workspace…** Menu, or use the **Manage Multi-Project Workspace…** button of the **Build Toolbar**.

Manage Multi-Project Workspace… – dialog to add individual projects or programs to your Multi-Project

Add existing stand-alone projects[1,2] to your Multi-Project. Use the controls to change the file order, to add or remove project files, or to define the active project.

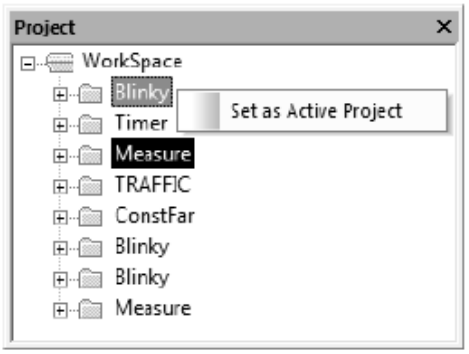
Removing or deleting a project from this list will not physically delete the project files, or the respective project from the storage location.



Activating a Multi-Project

To switch to another project, right click the project name you wish to activate, and click **Set as Active Project**.

In this example, *Measure* is the currently active project, whereas *Blinky* is just about to become the active project.



To uniquely identify the currently active project, µVision highlights its name in black. All actions executed within the µVision IDE apply only to this project; therefore, you can treat this project the same way you treat a stand-alone project.

---

[1] *Only existing projects can be maintained and added to a Multi-Project. You have to create the stand-alone project prior to managing it in the Multi-Project environment.*

[2] *Projects can have identical names as long as they reside in different folders.*

Batch-Building Multiple Projects

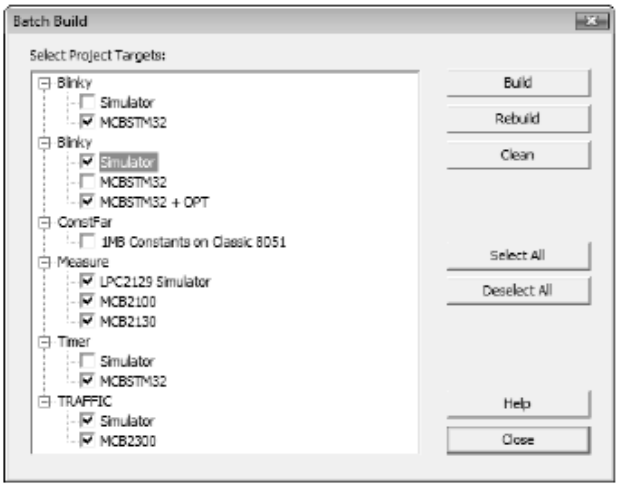
While you can compile the individual projects one-by-one, the Multi-Project environment provides a more convenient way to compile all the projects in one working step.

Use the **Batch Build**[1] command from the **Build Toolbar** or from the **Project – Batch Build** Menu to build, re-build, or clean the Project Targets.

Batch Build – opens the window which lets you select the targets and actions

Select the checkbox of the projects and related targets you wish to build, re-build, or clean.

Object files will be created based on the settings outlined in the respective project. No 'in common' object file will be created in addition.



The **Build** button compiles and assembles those files that have changed and links the selected targets.

The **Rebuild** button compiles or assembles all files and links the selected targets.

The **Clean** button removes the object files for the selected targets.

[1] *Batch Build can be used in a Multi-Project setup only.*

Chapter 7. Debugging

The µVision Debugger can be configured as a Simulator[1] or as a Target Debugger[2]. Go to the **Debug** tab of the **Options for Target** dialog to switch between the two debug modes and to configure each mode.

The **Simulator** is a software-only product that simulates most features of a microcontroller without the need for target hardware. By using the Simulator, you can test and debug your embedded application before any target hardware or evaluation board is available. µVision also simulates a wide variety of peripherals including the serial port, external I/O, timers, and interrupts. Peripheral simulation capabilities vary depending on the device you have selected.

The **Target Debugger** is a hybrid product that combines µVision with a hardware debugger interfacing to your target system. The following debug devices are supported:

- **JTAG/OCDS Adapters** that connect to on-chip debugging systems like the ARM Embedded ICE
- **Target Monitors** that are integrated with user hardware and that are available on many evaluation boards
- **Emulators** that connect to the MCU pins of the target hardware
- **In-System Debuggers** that are part of the user application program and provide basic test functions

Third-party tool developers may use the Keil Advanced GDI to interface µVision to their own hardware debuggers.

No matter whether you choose to debug with the Simulator or with a target debugger, the µVision IDE implements a single user interface that is easy to learn and master.

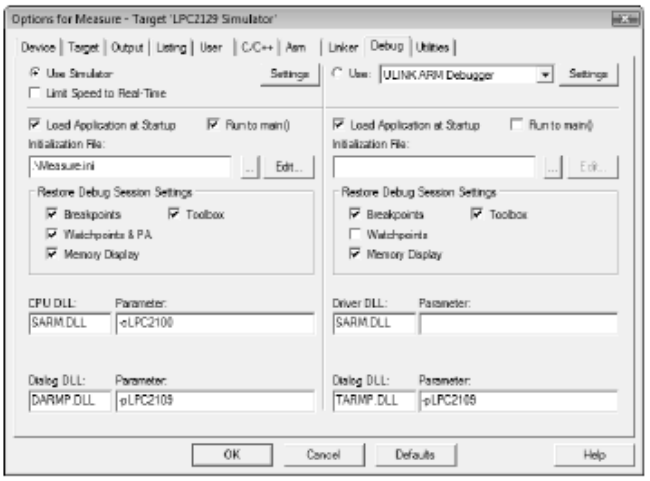
---

[1] *The Simulator offers more capabilities and features than those available when debugging on target hardware. The Simulator runs entirely on the PC and is not limited by hardware restrictions.*

[2] *Programs run on your target hardware. You can debug your application with restrictions.*

To debug programs using the Simulator, check **Use Simulator** on the left side of the **Debug** dialog.

To debug programs running on target hardware, check **Use *<Hardware Debugger>*** on the right side of the **Debug** dialog.



In addition to selecting whether you debug with the Simulator or Target Debugger, the **Debug** dialog provides a great variety of debugger configuration options.

Control	Description
Settings	Opens the configuration dialog for the simulation driver or the Advanced GDI target driver
Load Application at Startup	Loads the application program when you start the debugger
Limit Speed to Real-Time	Limits simulation speed to real-time such that the simulation does not run faster than the target hardware
Run to main()	Halts program execution at the main C function. When not set, the program will stop at an implicit breakpoint ahead of the main function
Initialization File	Specifies a command script file which is read and executed when you start the debugger, before program execution is started
Breakpoints	Restores breakpoint settings from the prior debug session
Watchpoints & PA	Restores watchpoints and Performance Analyzer settings from the prior debug session
Memory Display	Restores memory display settings from the prior debug session
Toolbox	Restores toolbox buttons from the prior debug session
CPU DLL	Specifies the instruction set DLL for the simulator. **Do not modify this setting.**
Driver DLL	Specifies the instruction set DLL for the target debugger. **Do not modify this setting.**
Dialog DLL	Specifies the peripheral dialog DLL for the simulator or target debugger. **Do not modify this setting.**

Simulation

µVision simulates up to 4 GB of memory from which specific areas can be mapped for reading, writing, executing, or a combination of these. In most cases, µVision can deduce the correct memory map from the program object module. Any illegal memory access is automatically trapped and reported.

A number of device-specific simulation capabilities are possible with µVision. When you select a microcontroller from the Device Database, µVision configures the Simulator accordingly and selects the appropriate instruction set, timing, and peripherals.

The µVision Simulator:

- Runs programs using the ARM7, ARM9, Thumb, Thumb2, 8051, C166/XE166/XC2000 instruction sets
- Is cycle-accurate and correctly simulates instructions and on-chip peripheral timing, where possible
- Simulates on-chip peripherals of many 8051, C166/XE166/XC2000, ARM7, ARM9, and Cortex-Mx devices
- Can provide external stimulus using the debugger C script language

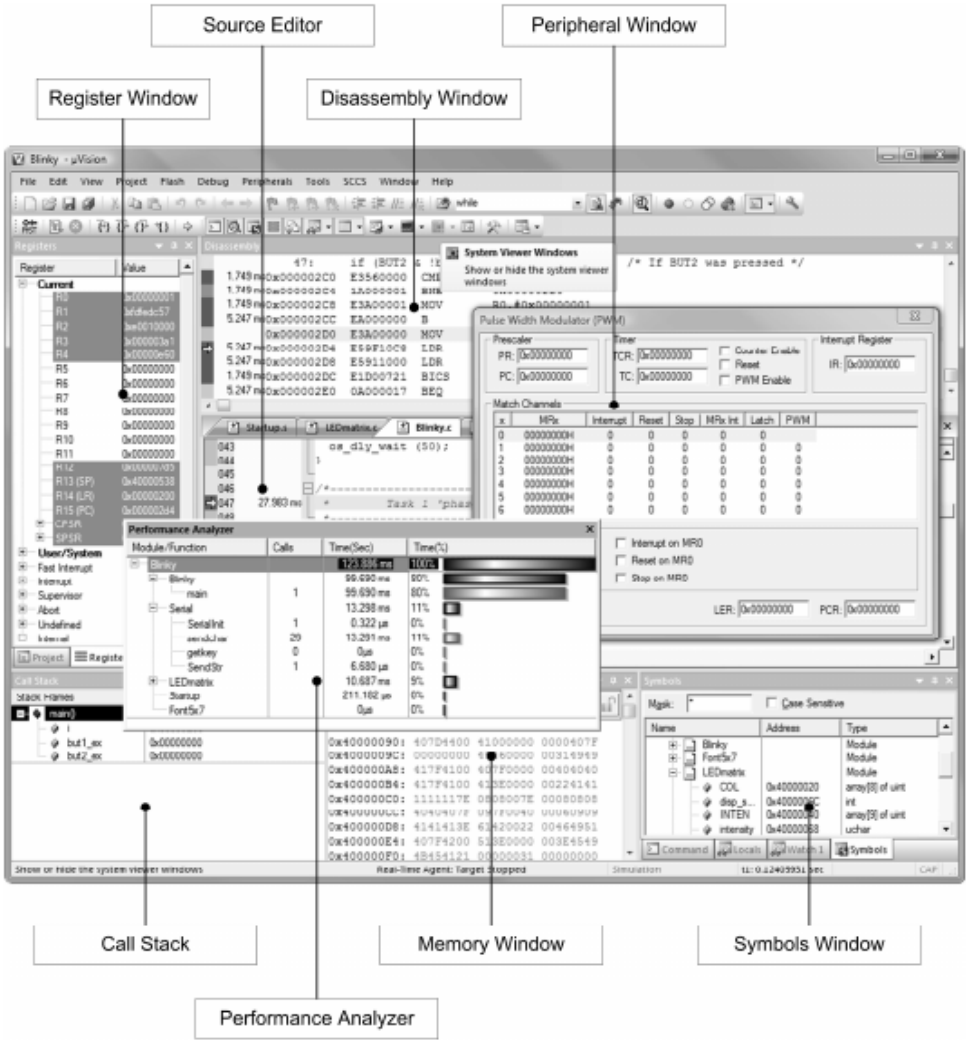
Starting a Debug Session

When you start a debug session, µVision loads the application, executes the startup code, and, if configured, stops at the main C function. When program execution stops, µVision opens a **Text Editor** window, with the current source code line highlighted, and a **Disassembly Window**, showing the disassembled code.

Use the **Start/Stop Debug Session** command of the **Debug Toolbar** to start or stop a debugging session. Screen layouts are restored when entering and saved automatically when closing the Debugger.

The current instruction or high-level statement (the one executed on the next instruction cycle) is marked with a yellow arrow. Each time you step, the arrow moves to reflect the new current line or instruction.

This screenshot below shows some of the key windows available in **Debug Mode**.



Debug Mode

Most editor features are also available while debugging. The **Find** command can be used to locate source text and source code can be modified. Much of the Debugger interface is identical to the Text Editor interface.

However, in **Debug Mode** the following additional features, menus, and windows are available:

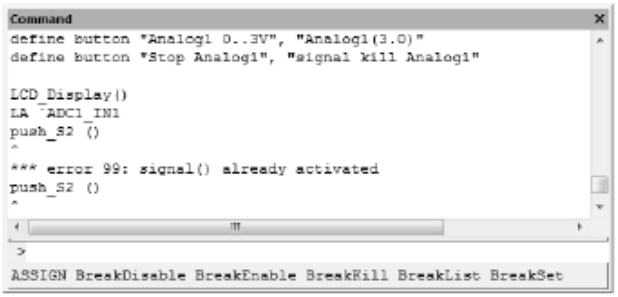
Debug Menu and **Debug Toolbar** – for accessing debug commands

- **Peripherals** Menu – is populated with peripheral dialogs used to monitor the environment
- **Command Window** – for executing debug commands and for showing debugger messages
- **Disassembly Window** – provides access to source code disassembly
- **Registers Window** – to view and change values in registers directly
- **Call Stack Window** – to examine the programs call tree
- **Memory**, **Serial**, and **Watch Windows** – to monitor the application
- **Performance Analyzer Window** – to fine tune the application for performance
- **Code Coverage Window** – to inspect the code for safety-critical systems
- **Logic Analyzer Window** – to study signals and variables in a graphical form
- **Execution Profiler** – to examine the execution time and number of calls
- **Instruction Trace Window** – to follow execution of the program sequence
- **Symbol Window** – to locate program objects comfortably
- **System Viewer** – to supervise peripheral registers
- **Multiple Debug Restore Layouts** – can be defined to switch between preferred window arrangements

Besides the disabled build commands, you may not:

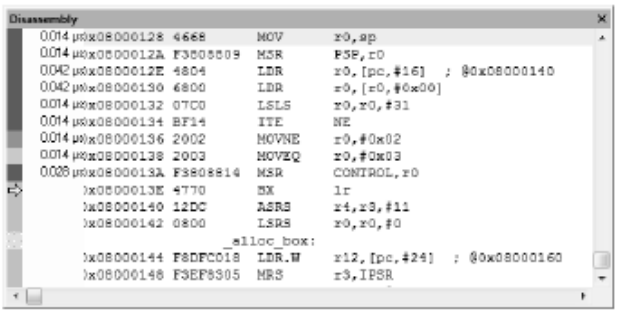
- Modify the project structure
- Change tool parameters

Using the Command Window



Generic compile and debug information are displayed here while stepping through the code. Additional notifications are provided if, for example, memory areas cannot be accessed. Enter debugger commands on the **Command Line** of the **Command Window**. Valid instructions will rise on its status bar with hints to parameters and parameter options. Insert expressions to view or modify the content of registers, variables, and memory areas. You can invoke debugger script functions as well. We strongly advise you to make use of the detailed on-line help information, by pressing **F1**. Describing the many options available is beyond the scope of this book.

Using the Disassembly Window



Configure this window by invoking its **Context Menu**. You can use this window to view the time an instruction needs to execute or to display the number of calls. You can also set or remove breakpoints and bookmarks.

View a trace history of previously executed instructions through the **View – Trace – View Trace Records** Menu. To view a history trace, enable the option **View – Trace – Enable Trace Recording**.

If the **Disassembly Window** is the active window, single-stepping works at the assembler instruction level rather than at the program source level.

Executing Code

µVision provides several ways to run your programs. You can

- Instruct the program to run directly to the main C function. Set this option in the **Debug** tab of the **Options for Target** dialog.
- Select debugger commands from the **Debug** Menu or the **Debug Toolbar**
- Enter debugger commands in the **Command Window**
- Execute debugger commands from an initialization file

Starting the Program

Select the **Run** command from
the **Debug Toolbar**
or **Debug** Menu
or type **GO** in the **Command Window** to run the program

Stopping the Program

Select **Stop** from
the **Debug Toolbar**
or from the **Debug** Menu
or press the **Esc** key while in the **Command Window**

Resetting the CPU

Select **Reset** from
the **Debug Toolbar**
or from the **Debug – Reset CPU** Menu
or type **RESET** in the **Command Window** to reset the simulated CPU

Single-Stepping

To step through the program and into function calls use the **Step** command from the **Debug Toolbar** or **Debug** Menu. Alternatively, you enter **TSTEP** in the **Command Window**, or press **F11**.

To step through the program and over function calls use the **Step Over** command from the **Debug Toolbar** or **Debug** Menu. Enter **PSTEP** in the **Command Window**, or press **F10**.

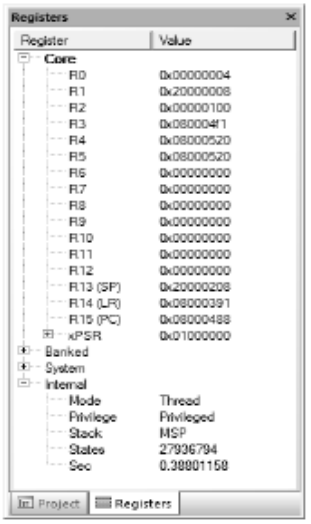
To step out of the current function use the **Step Out** command from the **Debug Toolbar** or **Debug** Menu. Enter **OSTEP** in the **Command Window**, or press **Ctrl+F11**.

Examining and Modifying Memory

µVision provides various ways to observer and change program and data memory. Several windows display memory contents in useful formats.

Viewing Register Contents

The **Registers Window** shows the content of microcontroller registers. To change the content of a register double-click on the value of the register. You may also press **F2** to edit the selected value.

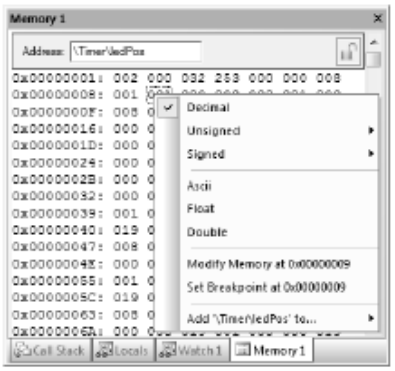


Memory Window

Monitor memory areas through four distinct **Memory Windows**.

Open the **Memory Window** from the **Debug Toolbar** or the **View – Memory – Memory[x]** Menu

The **Context Menu** allows you to select the output format.



Enter an expression in the **Address** field to monitor the desired area or object. To change the content of an address, double-click on the value and modify it.

To update the **Memory Window** periodically, enable **View – Periodic Window Update**. Use **Update Windows** in the **Toolbox** to refresh the windows manually.

To stop the **Memory Window** from refreshing, uncheck **View – Periodic Window Update**, or use the **Lock** button to get a snapshot of the window. You may compare values of the same address space by taking snapshots of the same section in a second **Memory Window**.

Memory Commands

The following memory commands can be entered in the **Command Window**.

Command	Description
ASM	Displays or sets the current assembly address and allows you to enter assembly instructions. When instructions are entered, the resulting op-code is stored in code memory. You may use the in-line assembler to correct mistakes or to make temporary changes to the target program.
DISPLAY	Displays a range of memory in the Memory Window (if it is open) or in the Command Window. Memory areas are displayed in HEX and in ASCII.
ENTER	Allows you to change the contents of memory starting at a specified address
EVALUATE	Calculates the specified expression and outputs the result in decimal, octal, HEX, and ASCII format
UNASSEMBLE	Disassembles code memory and displays it in the Disassembly Window

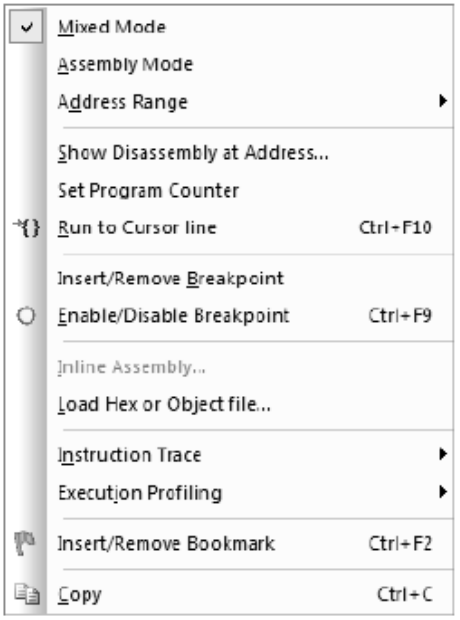
Breakpoints and Bookmarks

In µVision, you can set breakpoints and bookmarks while:

- Creating or editing your program source code
- Debugging, using the **Breakpoints** dialog, invoked from the **Debug** Menu
- Debugging, using commands you enter in the **Command Window**

Setting Breakpoints and Bookmarks

To set execution breakpoints in the source code or in the **Disassembly Window**, open the **Context Menu** and select the **Insert/Remove Breakpoint** command.



You can double-click the gray sidebar of the **Editor Window** or **Disassembly Window** to set a breakpoint, or use the breakpoint buttons of the **File Toolbar**.

Breakpoints and bookmarks visualize in the **Editor** and the **Disassembly Window** alike and differ in their coloring. Breakpoints will display in red, where as bookmarks can be recognized by their blue color.

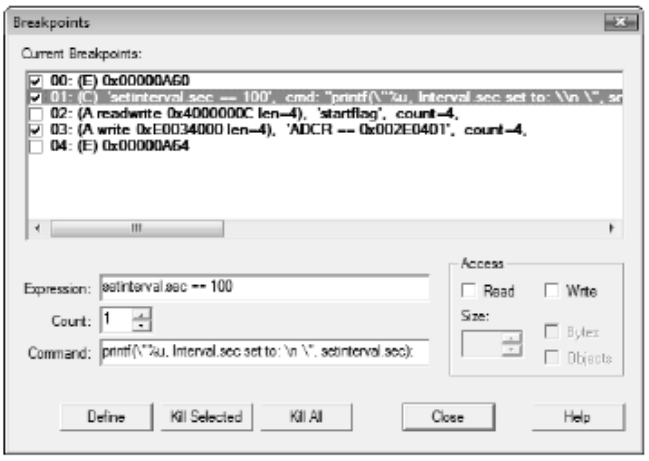
Analog actions are required to define bookmarks. In contrast to breakpoints, bookmarks will not stop the program executing.

Use **Bookmarks** to set reminders and markers in your source code. Define the critical spots easily and navigate quickly between bookmarks using the bookmark navigation commands. You can also define a bookmark and a breakpoint on the same line of code concurrently.

Whereas bookmarks do not require additional explanations, breakpoints are discussed in detail in the following section.

Breakpoints Window

Invoke the **Breakpoints Window** from the **Debug** Menu.



You have to stop the program running, to get access to this dialog.

Modify existing breakpoints and add new breakpoints via this dialog. Enable/disable breakpoints using the checkbox in the **Current Breakpoints** list. Double-click on an existing breakpoint to modify its definition.

Define a breakpoint by entering an **Expression**. Depending on the expression entered, one of the following breakpoint types is defined:

- An **Execution Breakpoint (E)** is defined when the expression specifies a code address. This breakpoint is triggered when the specified code address is reached. The code address must refer to the first byte of a microcontroller instruction.
- An **Access Breakpoint (A)** is defined when the expression specifies a memory access (read, write, or both) instruction. This breakpoint is triggered when the specified memory access occurs. You may specify the number of bytes or objects (based on the expression) which trigger the breakpoint. Expressions must reduce to a memory address and type. Operators (&, &&, <. <=. >, >=, = =, !=) may be used to compare values before the **Access Breakpoint** triggers and halts program execution or executes the **Command**.
- A **Conditional Breakpoint (C)** is defined when the expression specifies a true/false condition and cannot be reduced to an address. This breakpoint is triggered when the specified conditional expression is true. The conditional expression is recalculated after each instruction. Therefore, program execution may slow down considerably.

When a **Command** has been specified for a breakpoint, µVision executes the command and continues to execute your target program. The command specified can be a µVision debug function or signal function. To halt program execution in a µVision function, set the **_break_** system variable. For more information, refer to *System Variables* in the on-line help.

The **Count** value specifies the number of times the breakpoint expression is true before the breakpoint is triggered.

Breakpoint Commands

The following breakpoint commands can be entered in the **Command Windows**.

Command	Description
BREAKSET	Sets a breakpoint for the specified expression. Breakpoints are program addresses or expressions that, when true, halt execution of your target program or execute a specified command.
BREAKDISABLE	Disables a previously defined breakpoint
BREAKENABLE	Enables a previously defined breakpoint that is disabled
BREAKKILL	Removes a previously defined breakpoint
BREAKLIST	Lists all breakpoints

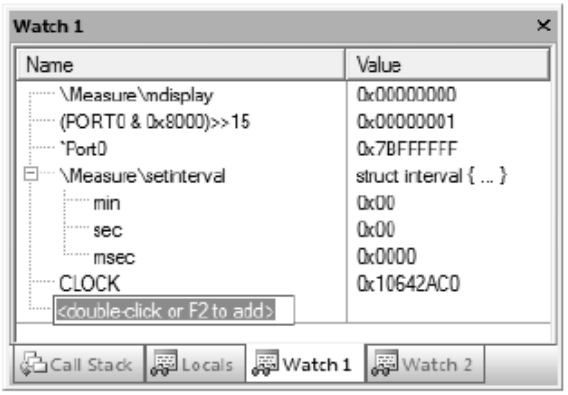
You may also set execution breakpoints while editing or debugging using buttons on the **File Toolbar**.

Watchpoints and Watch Window

By default, **Watch Windows** consist of four page tabs: the **Locals** to view variables of the current function, two **Watch** pages for personalized watchpoints, and the **Call Stack** showing the program tree. Through the **Watch Window**, you can view and modify program variables. Nested function calls are listed in this window as well. The content is updated automatically whenever you step through the code in **Debug Mode** and the option **View – Periodic Window Update** is set. In contrast to the **Locals Window**, which displays all local function variables, the **Watch Window** displays user-specific program variables.

Watchpoints

Define watchpoints to observe variables, objects, and memory areas affected by your target program. Watchpoints can be defined in two **Watch** pages. The **Locals Window** contains items of the currently executed function. Items are added automatically to the **Locals Window**.



There are several ways to add a watchpoint:

- In any **Watch Window**, use the field **<double-click or F2 to add>**
- Double-click an existing watchpoint to change the name
- In **Debug Mode**, open the **Context Menu** of a variable and use **Add *<item name>* to… – Watch Window**. µVision automatically selects the variable name beneath the mouse pointer. You can also mark an expression and add it to the **Watch Window**.
- In the **Command Window**, use the **WATCHSET** command to create a new watchpoint
- Finally, drag-and-drop any object from the **Symbols Window** or from source code files into the **Watch Window**

Modify local variables and watchpoint values by double-clicking the value you want to change, or click on the value and press **F2**. Remove a watchpoint by selecting it and press the **Del** key.

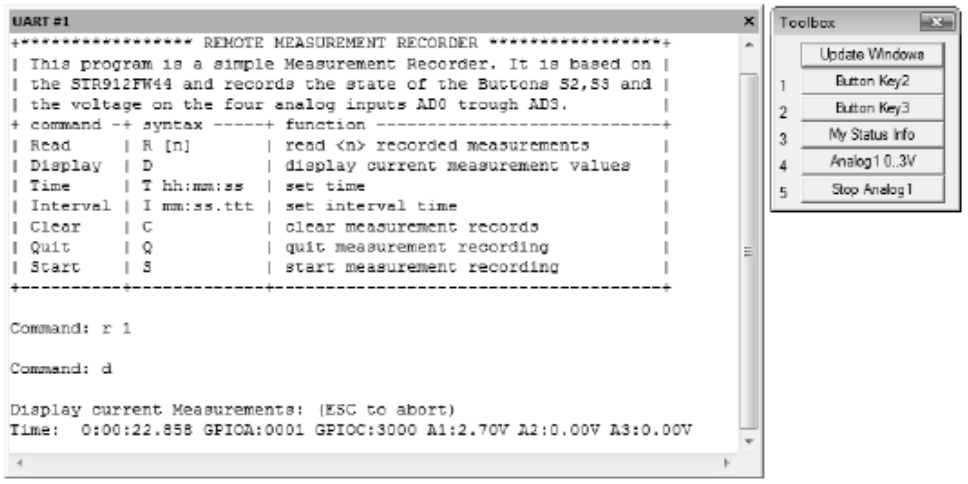
Watchpoint Commands

The following watchpoint commands can be entered in the **Command Window**.

Command	Description
WATCHSET	Defines a watchpoint expression to display in a Watch Window
WATCHKILL	Deletes all defined watchpoint expressions in any Watch Window

Serial I/O and UARTs

µVision provides three **Serial Windows**, named «**UART** #{1|2|3}», for each simulated on-chip UART. Serial data output from the simulated microcontroller are shown in these windows. Characters you type into the **Serial Window** are considered input to the simulated microcontroller.



The serial output can be assigned to a PC COM port using the **ASSIGN** Debugger command.

Several modes for viewing the data are provided:

- Basic VT100 Terminal Mode
- Mixed Mode
- ASCII Mode
- HEX Mode

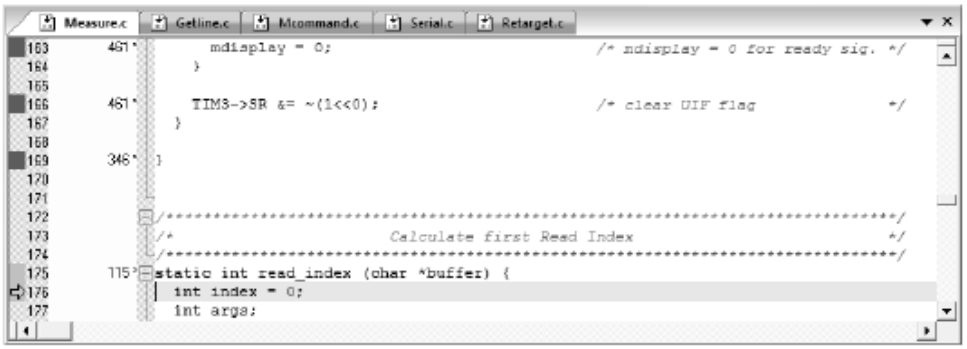
You can copy the content of the window to the clipboard or save it to a file. Where applicable, you can use the **Toolbox**[1] features to interact with the program.

[1] *You can add, remove, and change Toolbox buttons at any time. Use the Command Line in the Command Window for this purpose.*

Execution Profiler

The **Execution Profiler** in µVision records the amount of time and the number of times each assembler instruction and high-level statement in your program executes.

The amount of time and the number of calls, which are displayed in the **Disassembly Window** and in the **Editor Window** alike, are cumulative values.

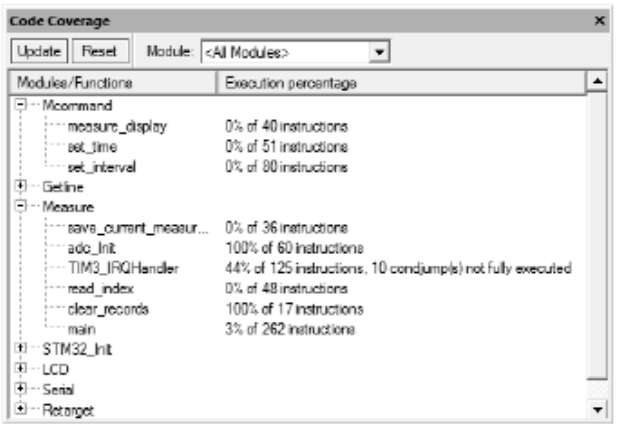


Enable the **Execution Profiler** through the **Debug – Execution Profiling** Menu.

Invoke the **Context Menu** of the **Disassembly Window** to switch between the time and calls.

When you locate program hotspots (with the **Performance Analyzer**), the **Execution Profiler** makes it easy to find real performance bottlenecks.

Code Coverage



The **Code Coverage Window** marks the code that has been executed, and groups the information based on modules and functions.

Use this feature to test safety-critical applications where certification and validation is required.

You can detect instructions that have been skipped, or have been executed fully, partially, or not at all.

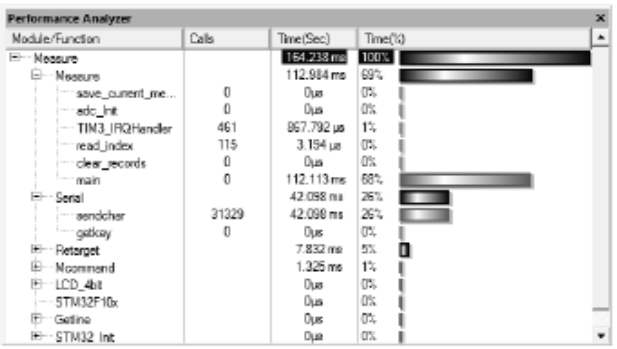
Code Coverage data can be saved to a file. You can even include a complete CPU instruction listing in this report. To make use of all these features, examine the **COVERAGE** command in the **Command Window**.

In addition to the **Code Coverage Window**, µVision provides color-coded hints on the side bar of the **Disassembly** and **Editor Window**. The colors have the following meaning:

- Lines not executed – are marked with a **grey** block
- Fully executed lines – are marked with a **green** block
- Skipped braches – are marked with an **orange** block
- Executed branches – are marked with a **blue** block
- Lines with no code – are marked with a **light grey checked** block

Performance Analyzer

The µVision **Performance Analyzer** displays the execution time recorded for functions in your application program.



Results show up as bar graphs along with the number of calls, the time spent in the function, and the percentage of the total time spent in the function.

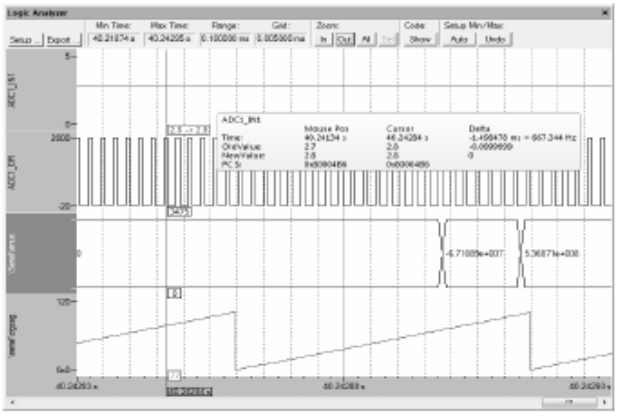
Use this information to determine where your program spends most of its time and what parts need further investigation.

Objects are sorted automatically dependent on the time spent.

Invoke the **Context Menu** of the **Performance Analyzer** to switch to another presentation of your investigation. You can drive the output to display statistics of modules or functions. Eventually, you might need to clean up the collected data to get a fresh summary.

Double-click an object exposed in the Module/Function column to jump to the source code line.

Logic Analyzer



The **Logic Analyzer** displays values of variables or virtual registers and shows the changes on a time axis.

Add values through the **Setup …** button or drag and drop objects from other windows into the **Logic Analyzer**. Press **Del**, or use the **Setup…** button, or invoke the **Context Menu** to remove items from the list.

The **Logic Analyzer** window contains several buttons and displays several fields to analyze data in detail. Move the mouse pointer to the desired location and wait one second to get additional information, which pops-up automatically.

Control	Description
Setup…	Define your variables and their settings through the **Logic Analyzer Setup** dialog
Export…	Saves the currently recorded signals to a tab-delimited file
Min Time	Displays the start time of the signal recording buffer
Max Time	Displays the end time of the signal recording buffer
Range	Displays the time range of the current display
Grid	Displays the time range of a grid line
Zoom	Changes the time range displayed. **Zoom All** shows the content of the buffer recording the signals. **Zoom Sel** zooms the display to the current selection (hold **Shift** and drag the mouse to mark a section).
Show	Opens the **Editor** or **Disassembly Window** at the code that caused the signal transition. It will also stop the program from executing.
Setup Min/Max	Configures the range of a signal. The **Auto** button configures the minimum and maximum values based on the values from the current recording. The **Undo** button restores the settings prior to using **Auto**.

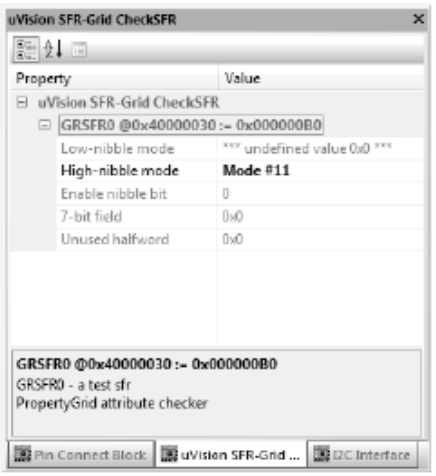
System Viewer

Peripheral Registers are memory mapped registers that a processor can write to and read from to control a peripheral device. µVision provides an advanced method for viewing and debugging these peripheral registers.

Invoke the **System Viewer** from the **Debug Toolbar** or from the **View – System Viewer Windows** Menu. You can define up to 100 different peripheral objects to monitor their behavior.

The **System Viewer** offers the following features:

- Parse a microcontroller device C header file into a binary format
- Additional properties can be added to the header file to provide extra information such as Peripheral Register descriptions and the data breakdown of an Peripheral Register
- The value of a Peripheral Register is updated either from the Simulator or from the target hardware. This can happen when the target is stopped, or periodically by enabling the **View — Periodic Window Update** Menu.
- At any time, the content of a Peripheral Register can be changed simply by overwriting its value in the **System Viewer**



Symbols Window

The **Symbols Window** displays information from the Debugger in an ordered and grouped manner and can be called via the **Debug Toolbar** or from the **View—Symbol Window** Menu. This functionality includes objects:

- Of simulated resources as the virtual registers, **Simulator VTREG**, with access to I/O pins, UART communication, or CAN traffic
- From Peripheral Registers, **Peripheral SFR**, to access peripherals
- Of the embedded application, recognizable by the name of the program, with access to functions, modules, variables, structures, and other source code elements

Use this functionality to find items quickly. Drag and drop the objects to any other window of µVision.

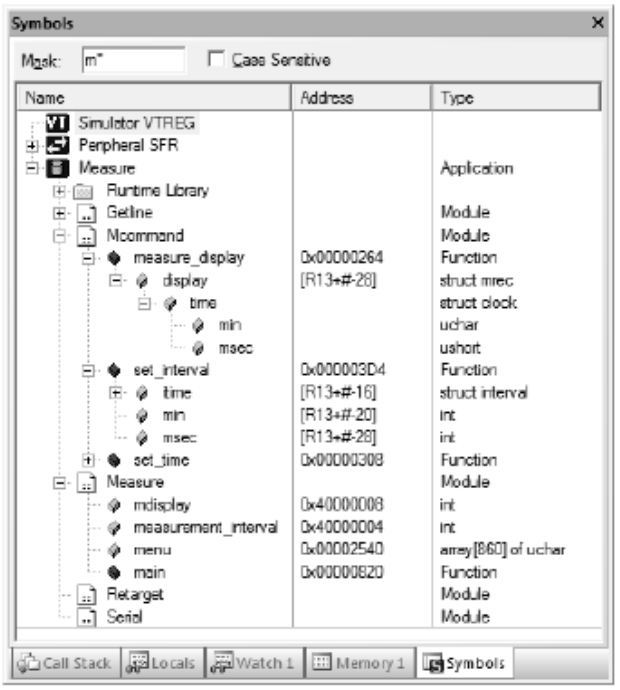
Mask works similar to a find function. Enter your search criteria and browse the results. For nested objects, the entire tree is displayed if a leaf item is found. The following search criteria may be used:

\# matches a digit (0 - 9)

$ matches any single character

* matches zero or more characters.

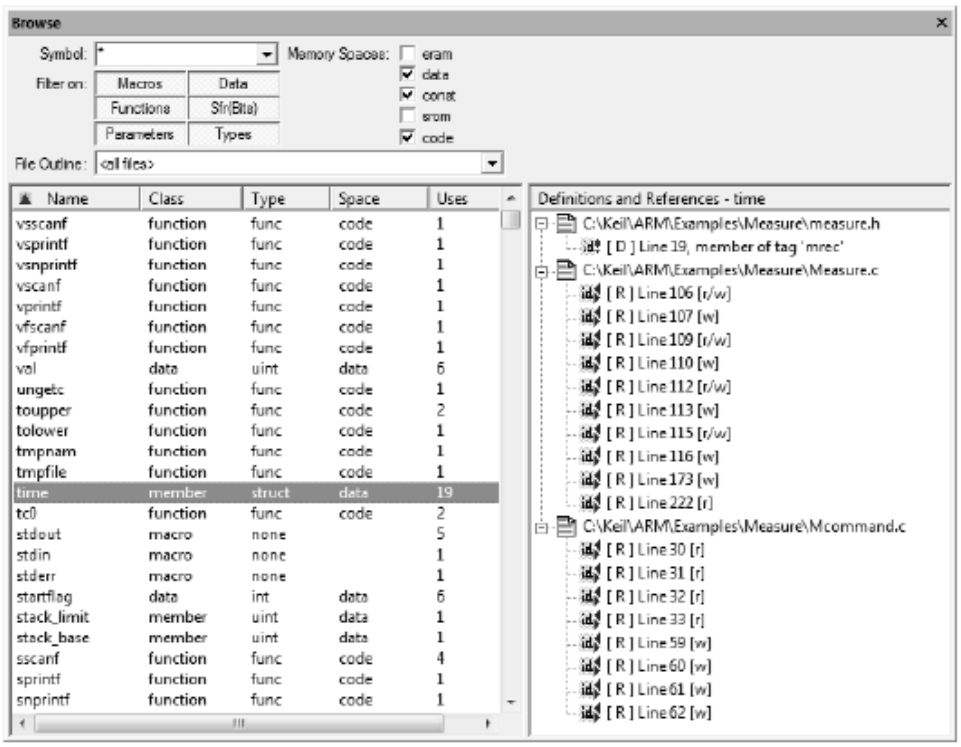
Configure the window by invoking the **Context Menu**.



Browse Window

The **Browse Window** enables you to search for objects in the code. This feature can be used in **Debug** and **Build Mode**. Nevertheless, the browse information is only available after compilation. You have to set the option **Options for Target – Output – Browser Information** to signal to the compiler to include browse information into the object file. Launch this window via the **File Toolbar** or **View – Source Browser Window**.

Enable or disable the **Filter on** buttons, enter your search criteria in the **Symbol** field and narrow the result through the **File Outline** drop-down. You can sort the results by clicking the header controls. Click an item to browse the occurrences and locate its usages. Double-click a line in the **Definition and References** page to jump to the code line.



Invoke the **Context Menu** while pointing at an item. Dependent on the object class you will get different options. For functions, you can invoke the callers graph and the call graph.

Toolbox

The **Toolbox** contains user-configurable buttons that execute debugger commands or user-defined functions. Click on a **Toolbox** button to execute the associated command. **Toolbox** buttons may be clicked at any time, even while the program executes.

Define a **Toolbox** button using the **Command Window** and the **DEFINE BUTTON**[1,2] command. Use the same command to redefine the button. The general syntax for this command is:

```
DEFINE BUTTON "button_label", "command"
```

Where:

`button_label` is the name that displays in the Toolbox

`command` is the command that executes when the button is pressed

The following examples show the commands used to create the buttons in the Toolbox shown above:

```
DEFINE BUTTON "Decimal Output", "radix=0x0A"
DEFINE BUTTON "My Status Info", "MyStatus ()"   /* call debug function  */
DEFINE BUTTON "Analog1 0..3V", "analog0 ()"     /* call signal function */

DEFINE BUTTON "Show R15", "printf (\"R15=%04XH\\n\")"
```

Remove a **Toolbox** button with the **KILL BUTTON**[3] command. The button number required in this statement is shown on the left side of the button. For example:

```
KILL BUTTON 5      /* resembles to: "Remove the 'Stop Analog1' button" */
```

---

[1] *The printf() command defined in the last example introduces nested strings. The double quote (") and backslash (\) characters of the format string must be escaped with \ to avoid syntax errors.*

[2] *Use this command to redefine the meaning of a button or change the description.*

[3] *The **Update Windows** button in the Toolbox is created automatically and cannot be removed. When pressed, this button updates the contents of several Debugger windows.*