KEil™
Tools by ARM

Getting Started
Creating Applications with µVision®4

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

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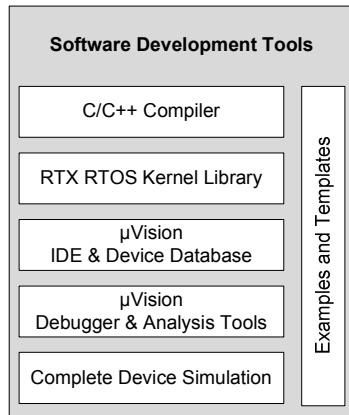
The following illustrations show the generic component blocks of µVision in conjunction with tools provided by Keil, or tools from other vendors, and the way the components relate.

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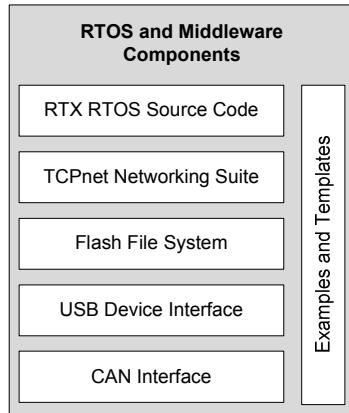
The RTX RTOS Kernel helps you to implement complex and time-critical software.

Software Development Tools	
C/C++ Compiler	
RTX RTOS Kernel Library	Examples and Templates
μVision IDE & Device Database	
μVision Debugger & Analysis Tools	
Complete Device Simulation	

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.

RTOS and Middleware Components	
RTX RTOS Source Code	
TCPnet Networking Suite	Examples and Templates
Flash File System	
USB Device Interface	
CAN Interface	

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^{2}C, 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.*

C166, XE166 and XC2000 Architecture Advantages

- Extremely fast I/O operations via the Peripheral Event Controller

- High-speed interrupt system with very well-tuned peripherals

- Efficient arithmetic and fast memory access

These devices are used in medium to large sized applications that require high I/O throughput. This architecture is well suited to the needs of embedded systems that involve a mixture of traditional controller code and DSP algorithms.

ARM7 and ARM9 Architecture Advantages

- Huge linear address space

- The 16-bit Thumb instruction set provides high code density

- Efficient support for all C integer data types including pointer addressing

ARM7 and ARM9-based microcontrollers are used for applications with large memory demands and for applications that use PC-based algorithms.
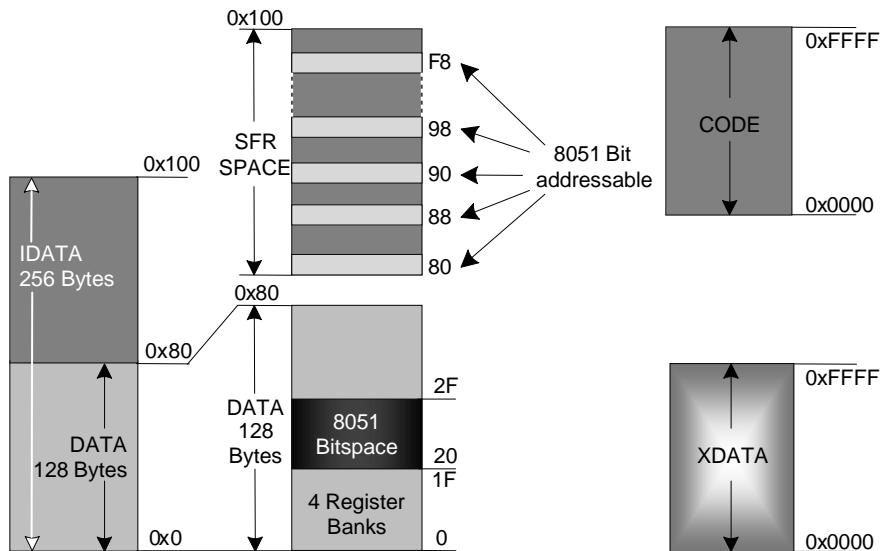
Cortex-Mx Architecture Advantages

- One instruction set, Thumb2, reduces the complexity of the program code and eliminates the overhead needed for switching between ARM and Thumb instruction mode

- The Nested Vector Interrupt Controller (NVIC) removes interrupt prolog and epilog code, and provides several, configurable priority levels

- Extremely low power consumption with a variety of sleep modes

The Cortex-Mx microcontroller architecture is designed for hard real-time systems, but can be used for complex System-on-Chip applications as well.

Classic and Extended 8051 Devices

8051 devices combine cost-efficient hardware with a simple but efficient programming model that uses various memory regions to maximize code efficiency and speed-up memory access. The following figure shows the memory layout of a classic 8051 device.

The 8051 architecture provides three different physical memory regions:

- **DATA/IDATA** memory includes a 256 Bytes on-chip RAM with register banks and bit-addressable space that is used for fast variable accessing. Some devices provide an extended data (**EDATA**) space with up to 64KB.

- **CODE** memory consists of 64KB ROM space used for program code and constants. The Keil linker supports code banking that allows you to expand the physical memory space. In extended variants, up to 16MB ROM space is available.

- **XDATA** memory has a 64KB RAM space for off-chip peripheral and memory addressing. Today, most devices provide some on-chip RAM that is mapped into **XDATA**.

- **SFR** and **IDATA** memory are located in the same address space but are accessed through different assembler instructions

- For extended devices, the memory layout provides a universal memory map that includes all 8051-memory types in a single 16MByte address region

8051 Highlights

- Fast interrupt service routines with two or four priority levels and up to 32-vectored interrupts

- Four register banks for minimum interrupt prolog/epilog

- Bit-addressable space for efficient logical operations

- 128 Bytes of Special Function Register (SFR) space for tight integration of on-chip peripherals. Some devices extend the SFR space using paging.

- Low-power, high-speed devices up to 100 MIPS are available

8051 Development Tool Support

The Keil C51 Compiler and the Keil Linker/Locator provide optimum 8051 architecture support with the following features and C language extensions.

- Interrupt functions with register bank support are written directly in C

- Bit and bit-addressable variables for optimal Boolean data type support

- Compile-time stack with data overlaying uses direct memory access and gives high-speed code with little overhead compared to assembly programming

- Reentrant functions for usage by multiple interrupt or task threats

- Generic and memory-specific pointers provide flexible memory access

- Linker Code Packing gives utmost code density by reusing identical program sequences

- Code and Variable Banking expand the physical memory address space

- Absolute Variable Locating enables peripheral access and memory sharing

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:
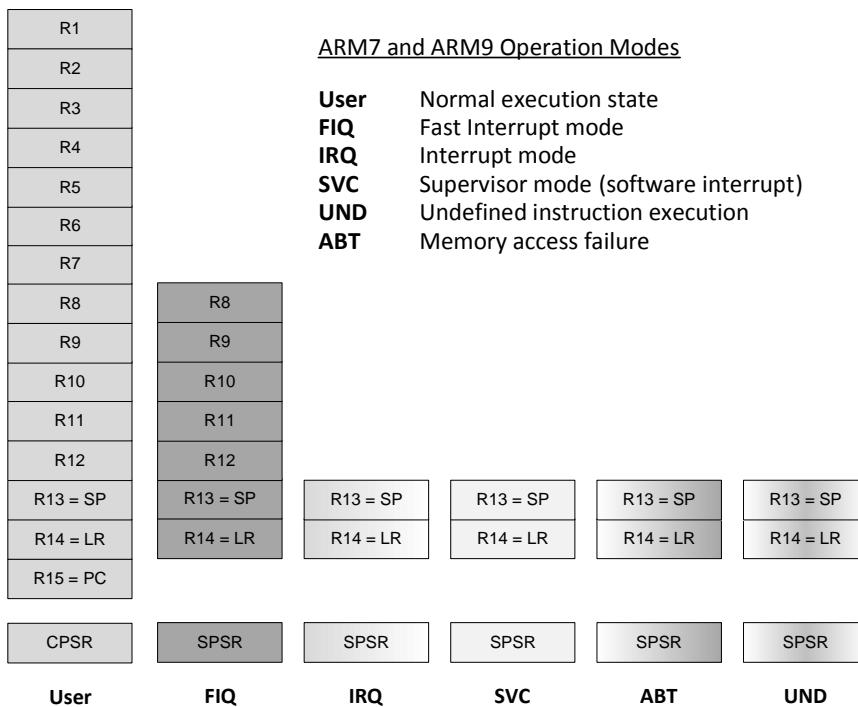
- Memory type support and flexible digital pattern processing for extremely fast variable access

- 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.

R1

R2

R3

R4

R5

R6

R7

ARM7 and ARM9 Operation Modes

User	Normal execution state
FIQ	Fast Interrupt mode
IRQ	Interrupt mode
SVC	Supervisor mode (software interrupt)
UND	Undefined instruction execution
ABT	Memory access failure

R8	R8
R9	R9
R10	R10
R11	R11
R12	R12
R13 = SP	R13 = SP	R13 = SP	R13 = SP	R13 = SP	R13 = SP
R14 = LR	R14 = LR	R14 = LR	R14 = LR	R14 = LR	R14 = LR
R15 = PC

CPSR	SPSR	SPSR	SPSR	SPSR	SPSR
User	**FIQ**	**IRQ**	**SVC**	**ABT**	**UND**

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.

ARM7 and ARM9 Development Tool Support

The ARM compilation tools support all ARM-specific features and provide:
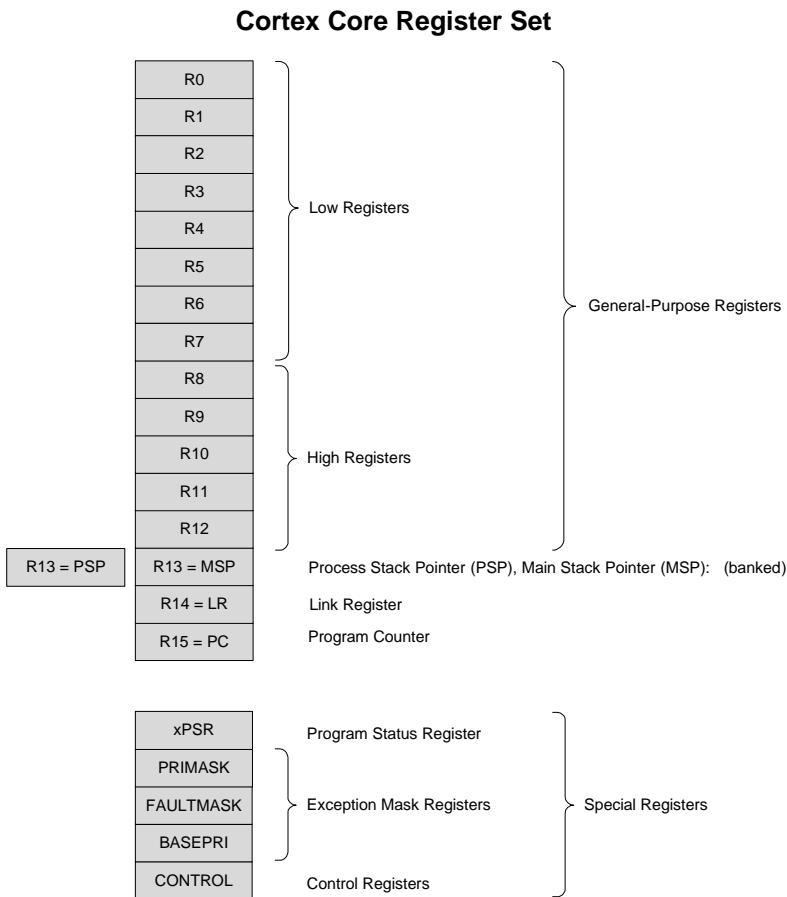
- **Function Inlining** eliminates call/return overhead and optimizes parameter passing

- **Inline assembly** supports special ARM/Thumb instructions in C/C++ programs

- **RAM functions** enable high-speed interrupt code and In-System Flash programming

- **ARM/Thumb interworking** provides outstanding code density and microcontroller performance

- **Task function and RTOS support** are built into the C/C++ compiler

Cortex-Mx based Microcontrollers

Designed for the 32-bit microcontroller market, the Cortex-Mx microcontrollers combine excellent performance at low gate count with features only previously found in high-end processors.

With 4GB of linear, unified memory space, the Cortex-Mx processors provide bit-banding features and supports big and little endian configuration. Predefined memory types are available, while some memory regions have additional attributes. Code can be located in the SRAM, external RAM, but preferably in the Code region. Peripheral registers are mapped into the memory space. Code density is improved by the Thumb or Thumb2 instruction set, depending on the processor version.

General-purpose registers rank from R0 to R12. R13 (SP) is banked, with only one copy of the R13 (MSP, PSP) being visible at a time. Special registers are available, but are not used for normal data processing. Some of the 16-bit Thumb instructions can access R0-R7 (low) registers only. There is no FIQ; however, nested interrupts and interrupt priority handling is implemented via the Nested Vector Interrupt Controller (NVIC), greatly reducing interrupt latency.

Cortex Core Register Set

R0	⎫
R1	
R2	
R3	
R4	Low Registers
R5	
R6	
R7	
R8	⎫
R9	
R10	High Registers
R11	
R12	⎭

General-Purpose Registers

R13 = PSP	R13 = MSP	Process Stack Pointer (PSP), Main Stack Pointer (MSP): (banked)
	R14 = LR	Link Register
	R15 = PC	Program Counter

xPSR	Program Status Register
PRIMASK	
FAULTMASK	Exception Mask Registers
BASEPRI	
CONTROL	Control Registers

Special Registers

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**

Generating Optimum Code

The C/C++ compilers provided by Keil are leaders in code generation and produce highly efficient code. However, code generation and translation is influenced by the way the application software is written. The following hints will help you optimize your application performance.

Coding Hints for All Architectures

Hint	Description
Keep **interrupt** functions short.	Well-structured interrupt functions only perform data collection and/or time-keeping. Data processing is done in the main function or by RTOS task functions. This reduces overhead involved with context save/restore of interrupt functions.
Check the requirement for **atomic operations**.	Atomic code is required for accessing data while using multiple RTOS threads or interrupt routines that access the memory used by the main function. Carefully check the application to determine if atomic operations are needed and verify the generated code. The various architectures have different pitfalls. For example, incrementing a variable on the 8051 and C166/XE166/XC2000 device is a single, atomic instruction, since it cannot be interrupted, whereas multiple instructions are required for an increment on ARM devices. In contrast, the 8051 requires multiple instructions to access the memory of an **int** variable.
Apply the **volatile** attribute on variables that are modified by an interrupt, hardware peripherals, or other RTOS tasks.	The **volatile** attribute prevents the C/C++ compiler from optimizing variable access. By default, a C/C++ Compiler may assume that a variable value will remain unchanged between several memory-read operations. This may yield incorrect application behavior in real-time applications.
When possible, use **automatic** variables for loops and other temporary calculations.	As part of the optimization process, the Keil C/C++ compiler attempts to maintain local variables (defined at function level) in CPU registers. Register access is the fastest type of memory access and requires the least program code.

Coding Hints for the 8051 Architecture

Hint	Description
Use the smallest possible data type for variables. Favor **unsigned char** and **bit**.	The 8051 uses an 8-bit CPU with extensive bit support. Most instructions operate on 8-bit values or bits. Consequently, small data types generate code that is more efficient.
Use **unsigned** data types whenever possible.	The 8051 has no direct support for signed data types. Signed operations require additional instructions whereas unsigned data types are directly supported by the architecture.
Favor the **SMALL** memory model.	Most applications may be written using the **SMALL** memory model. You can locate large objects, as arrays or structures, into **xdata** or **pdata** memory using explicit memory types. Note, the Keil C51 run-time library uses generic pointers and can work with any memory type.
When using other memory models, apply the memory type **data** to frequently used variables.	Variables in the **data** address space are directly accessed by an 8-bit address that is encoded into the 8051 instruction set. This memory type generates the most efficient code.
Learn how to use **pdata** memory type on your device.	The **pdata** memory provides efficient access to 256 bytes using MOVX @Ri instructions with 8-bit addressing. However, **pdata** behaves differently on the various 8051 devices, since it may require setting up a paging register. The **xdata** memory type is generic and accesses large memory spaces (up to 64KB).
Use **memory-typed pointers** when possible.	By default, the Keil C51 Compiler uses generic pointers that may access any memory type. Memory-typed pointers can access only a fixed memory space, but generate faster and smaller code.
Reduce the usage of **Reentrant Functions**.	The 8051 lacks support for stack variables. Reentrant functions are implemented by the Keil C51 Compiler using a compile-time stack with data overlaying for maximum memory utilization. Reentrant functions on the 8051 require simulation of the stack architecture. Since reentrant code is rarely needed in embedded applications, you should minimize the usage of the reentrant attributes.
Use the **LX51 Linker/Locater** and **Linker Code Packing** to reduce program size.	The extended LX51 Linker/Locator (available only in the PK51 Professional Developer's Kit) analyzes and optimizes your entire program. Code is reordered in memory to maximize 2-byte **AJMP** and **ACALL** instructions (instead of 3-byte **LJMP** and **LCALL**). **Linker Code Packing** (enabled in C51 OPTIMIZE level 8 and above) generates subroutines for common code blocks.

Coding Hints for C166, XE166, XC2000 Architectures

Hint	Description
When possible, use 16-bit data types for **automatic** and **parameter** variables.	Parameter passing is performed in 16-bit CPU registers (many 16-bit registers are available for automatic variables). More 16-bit variables (**signed/unsigned int/short**) can be assigned to CPU registers. This generates code that is more efficient.
Replace **long** with **int** data types when possible.	Operations that use 16-bit types (like **int** and **unsigned int**) are much more efficient than operations using **long** types.
Use the **bit** data type for boolean variables.	These CPUs have efficient bit instructions that are fully supported by the Keil C166 Compiler with the **bit** data type.
Use the **SMALL** or **MEDIUM** memory model when possible.	In these memory models, the default location of a variable is in **near** memory, accessible through16-bit direct addresses encoded in the CPU instructions. You can locate large objects (array or struct) into **huge** or **xhuge** using explicit memory types.
When using other memory models, apply the **near**, **idata**, or **sdata** memory type to frequently used variables.	Variables in the **near**, **idata**, or **sdata** address space are accessed through a 16-bit address that is encoded directly into a single C166/XE166/XC2000 instruction. These memory types generate the most efficient code.
Use the memory model **HCOMPACT/HLARGE** instead of **COMPACT/LARGE**.	The memory models COMPACT and LARGE use the obsolete **far** memory type and have an object size limit of 16KB. The memory models HCOMACT and HLARGE use the **huge** memory type that feature a 64KB object size limit. Even cast operations from **near** to **huge** pointers are more optimal.
Use **near pointers** when possible.	Check if a **near** pointer is sufficient for accessing the memory, since **near** pointers can access variables in the **near**, **idata**, or **sdata** address space. **Near** pointers generate faster and smaller code.

Coding Hints for the ARM7 and ARM9 Architecture

Hint	Description
When possible, use 32-bit data types for **automatic** and **parameter** variables.	Parameter passing is performed in 32-bit CPU registers. All ARM instructions operate on 32-bit values. In Thumb mode, all stack instructions operate only on 32-bit values. By using 32-bit data types (**signed/unsigned int/long**), additional data type cast operations are eliminated.
Use the **Thumb** instruction set.	Thumb mode is about 65% of the code size and 160% faster than ARM mode when executing from a 16-bit memory system. The MDK-ARM Compiler automatically inserts required ARM / Thumb interworking instructions.
Use __**swi** software interrupt functions for atomic sequences.	Via the __swi function attribute, the MDK-ARM Compiler offers a method to generate software interrupt functions directly, which cannot be interrupted by IRQ (__swi functions can be interrupted by FIQ interrupts). In contrast to other embedded architectures, ARM prevents access to the interrupt disable bits **I** and **F** in **User mode**.
Enhance struct pointer access by placing **scalars at the beginning** and **arrays as subsequent struct** members.	Thumb and ARM instructions encode a limited displacement for memory access. When a struct is accessed via a pointer, scalar variables at the beginning of a struct can be accessed directly. Arrays always require address calculation. Consequently, it is more efficient to place scalar variables at the beginning of a struct.
Assign high speed interrupt code to RAM.	Code executed from Flash ROM typically requires wait states or CPU stalls. Code execution from RAM does not. Consequently, time critical functions (like high-speed interrupt code) can be located in RAM directly using the **Memory Assignment** feature in **Options for File – Properties** available via the **Context Menu** of that file.
Optimize for Size	To optimize an application for minimal program size select under **Options for Target** the following toolchain: ▪ In the dialog page Target enable Code Generation - Use Cross-Module Optimization ▪ In the dialog page **C/C++** select Optimization: Level 2 (-O2) and disable the options **Optimize for Time**, **Split Load and Store Multiple**, and **One ELF Section per Function**
MicroLIB	The compiler offers a MicroLIB to be used for further **reducing** the **code size** of an application. MicroLIB is tailored for deeply embedded systems, but is not fully ANSI compliant. Do not use MicroLIB when execution speed is your primary goal.
Optimize for Speed	To optimize an application for maximum execution speed, under **Options for Target** select the following toolchain: ▪ In the dialog page **Target** enable **Code Generation - Use Cross-Module Optimization** ▪ In the dialog page **C/C++** select Optimization: Level 3 (-O3), enable **Optimize for Time**, and disable **Split Load and Store Multiple**

Coding Hints for the Cortex-Mx Architecture

Hint	Description
When possible, use 32-bit data types for **automatic** and **parameter** variables.	Parameter passing is performed in 32-bit CPU registers. All ARM instructions operate on 32-bit values. In Thumb mode, all stack instructions operate only on 32-bit values. By using 32-bit data types (**signed/unsigned int/long**), additional data type cast operations are eliminated.
Optimize for Size	To optimize an application for minimal program size select under **Options for Target** the following toolchain: ▪ In the dialog page Target enable Code Generation - Use Cross-Module Optimization ▪ In the dialog page **C/C++** select Optimization: Level 2 (-O2) and disable the options **Optimize for Time**, **Split Load and Store Multiple**, and **One ELF Section per Function**
MicroLIB	The compiler offers a MicroLIB to be used for further **reducing** the **code size** of an application. MicroLIB is tailored for deeply embedded systems, but is not fully ANSI compliant. Do not use MicroLIB when execution speed is your primary goal.
Optimize for Speed	To optimize an application for maximum execution speed, under **Options for Target** select the following toolchain: ▪ In the dialog page**Target** enable **Code Generation - Use Cross-Module Optimization** ▪ In the dialog page **C/C++** select Optimization: Level 3 (-O3), enable **Optimize for Time**, and disable **Split Load and Store Multiple**
Sleep mode features	To optimize power consumption of an application you may use the `WFI` instruction to send the processor into Sleep Mode until the next interrupt is received. In C programs, use the intrinsic function __**wfi()** to insert this instruction into your code.
Enhance `struct` pointer access, by placing **scalars at the beginning** and **arrays as sub-sequent struct** members.	Thumb2 instructions encode a limited displacement for memory access. When a `struct` is accessed via a pointer, scalar variables at the beginning of a `struct` can be directly accessed. Arrays always require address calculations. Therefore, it is more efficient to place scalar variables at the beginning of a `struct`.
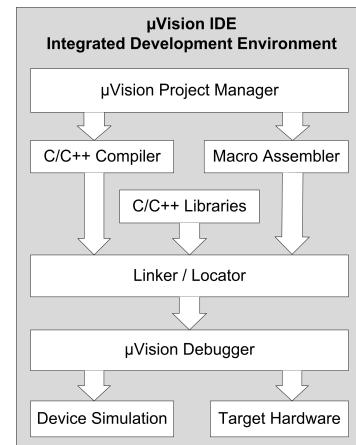
Chapter 3. Development Tools

The Keil development tools offer numerous features and advantages that help you to develop embedded applications quickly and successfully. They are easy to use and are guaranteed to help you achieve your design goals in a timely manner.

Software Development Cycle

When using the Keil tools, the project development cycle is similar to any other software development project.

1. Create a project, select the target device from the Device Data base, and configure the tool settings

2. Create your source files in C/C++ or Assembly

3. Build your application with the Project Manager

4. Debug and correct errors in source files, verify and optimize your application

5. Download your code to Flash ROM or SRAM and test the linked application

Each component shown in the block diagram is described in the following section.

µVision IDE

The µVision IDE is a window-based software development platform combining a robust editor, Project Manager, and Make Utility tool. µVision supports all the Keil tools including C/C++ Compiler, Macro Assembler, Linker, Library Manager, and Object-HEX Converter. µVision helps expedite the development process by providing:

- **Device Database** for selecting a device and configuring the development tools for that particular microcontroller

- **Project Manager** to create and maintain projects

- **Make Utility** for assembling, compiling, and linking your embedded applications

- Full-featured source code editor

- **Template Editor** that is used to insert common text sequences or header blocks

- **Source Browser** for rapidly exploring code objects, locating and analyzing data in your application

- **Function Browser** for quickly navigating between functions in your program

- Function Outlining for controlling the visual scope within a source file

- Built-in utilities, such as **Find in Files** and functions for commenting and uncommenting source code

- µVision **Simulator** and **Target Debugge**r are fully integrated

- **Configuration Wizard** providing graphical editing for microcontroller startup code and configuration files

- Interface to configure **Software Version Control Systems** and third-party utilities

- **Flash Programming Utilities**, such as the family of Keil ULINK USB-JTAG Adapters

- **Dialogs** for all development tool settings

- **On-line Help** and links to microcontroller data sheets and user guides

µVision Device Database

The µVision Device Database offers a convenient way to select and configure your device and project parameters. It includes preconfigured settings, so that you can fully concentrate on your application requirements. In addition, you can add your own devices, or change existing settings. Use the features of the Device Database to:

- Initialize the start up code and device settings
- Load the configuration options for the assembler, compiler, and linker
- You can add and change microcontroller configuration settings

µVision Debugger

The µVision Debugger is completely integrated into the µVision IDE. It provides the following features:

- **Disassembly** of the code on C/C++ source- or assembly-level with program execution in various **stepping modes** and various **view modes**, like assembler, text, or mixed mode
- Multiple **breakpoint** options including access and complex breakpoints
- **Bookmarks** to quickly find and define your critical spots
- **Review** and **modify** memory, variable, and register values
- **List** the **program call tree** including stack variables
- **Review** the status of on-chip microcontroller **peripherals**
- **Debugging commands** or **C-like scripting** functions
- **Execution Profiling** to record and display the time consumed, as well as the cycles needed for each instruction
- **Code Coverage** statistics for **safety-critical** application **testing**
- Various **analyzing tools** to view statistics, record values of variables and peripheral I/O signals, and to display them on a time axis
- **Instruction Trace** capabilities to view the **history** of executed instructions
- Define **personalized** screen and window **layouts**

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 re-locatable 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**.

Chapter 4. RTX RTOS Kernel

This chapter discusses the benefits of using a Real-Time Operating System (RTOS) and introduces the features available in Keil RTX Kernels. Note that the Keil development tools are compatible with many third-party RTOS solutions. You are not bound to use Keil RTX; however, the RTX Kernels are well integrated into the development tools and are feature-rich, and well tailored towards the requirements of deeply embedded systems.

Software Concepts
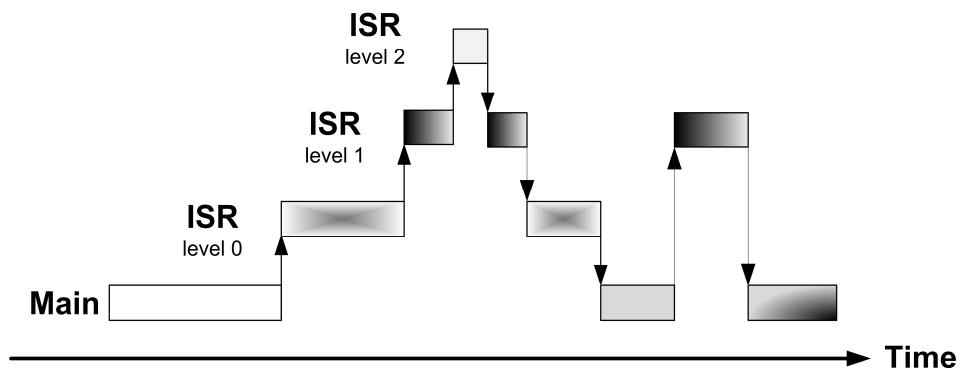
There are two basic design concepts for embedded applications:

- **Endless Loop Design:** this design involves running the program as an endless loop. Program functions (tasks) are called from within the loop, while interrupt service routines (ISRs) perform time-critical jobs including some data processing.

- **RTOS Design:** this design involves running several tasks with a **Real-Time Operating System (RTOS)**. The RTOS provides inter-task communication and time management functions. A preemptive RTOS reduces the complexity of interrupt functions, since time-critical data processing is performed in high-priority tasks.

Endless Loop Design

Running an embedded program in an endless loop is an adequate solution for simple embedded applications. Time-critical functions, typically triggered by hardware interrupts, are executed in an ISR that also performs any required data processing. The main loop contains only basic operations that are not time-critical, but which are executed in the background.

This software concept requires only one stack area and is very well suited for devices with limited memory. Architectures that provide several interrupt levels allow complex low-level ISR functions. Time-critical jobs may execute in higher interrupt levels.

8051, C166/XE166/XC2000, and ARM Cortex-Mx microcontrollers provide several interrupt levels. Higher-level interrupts may halt lower-level interrupts, or the main function.
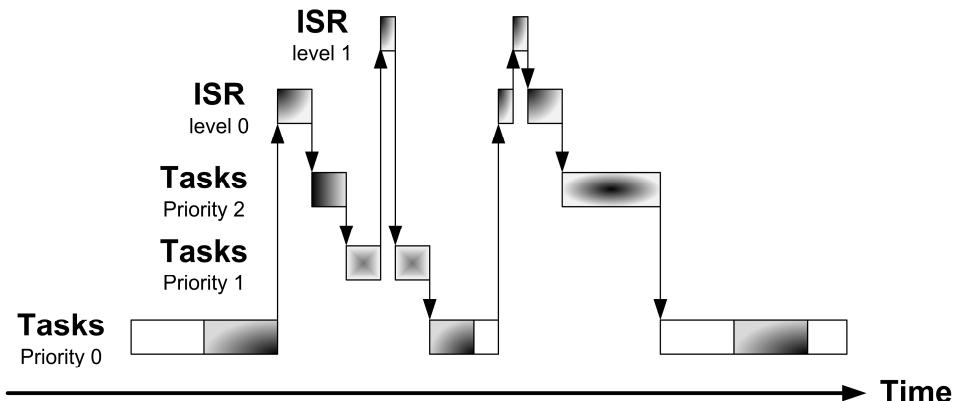
It is impossible to suspend the execution of an ISR except through higher priority interrupts. Therefore, the timing of a system with many complex ISR levels is unpredictable, since high priority interrupts may take up most of the CPU time.

Another challenge is to determine the worst-case stack nesting. Applications with complex ISR designs can have unnoticed stack resource issues, which may cause sporadic execution faults. Note, that the ARM architecture provides an extra stack for ISR that avoids stack memory surprises during the main loop execution.

RTOS Design

The RTOS design, due to its very nature, allows several tasks to execute within sequential time slices. A preemptive RTOS provides task priority levels, in which high priority tasks interrupt the execution of low priority tasks. Most RTOS systems offer inter-task communication and time delay functions supporting the design of complex applications.

The ARM based architectures are designed for RTOS usage. An RTOS is almost mandatory on ARM7, ARM9, and Cortex-Mx based systems that have several interrupt sources. ARM devices provide a separate ISR stack, and hence, each task needs no additional stack for ISR execution (as required on 8051 and C166/XE166/XC2000 devices).

A preemptive RTOS supports multiple task priorities. Tasks with the same priority are executed in sequence; tasks with a higher priority suspend tasks with a lower priority. An ISR always interrupts task execution and may exchange data with other tasks.

The RTOS also solves many other challenges specific to embedded applications. It helps you to maintain memory resources and data communication facilities, and allows you to split a complex application into simpler jobs.

Keil provides several different RTOS systems for the various microcontroller architectures:

- **RTX51 Tiny** and **RTX166 Tiny** (for 8051 and C166/XE166/XC2000) is a non-preemptive RTOS and uses a special stack swapping technique designed for devices with limited RAM

- **RTX** (for ARM7/ARM9 and Cortex-Mx) and **ARTX166** (for C166/XE166/XC2000) are preemptive RTOS kernels offering task priority levels. These kernels support message passing with ISRs and implement functions with thread-safe memory block allocation and deterministic execution times. An ISR may collect data into message buffers and send messages to a high priority task, which subsequently performs complex data processing. The ISR remains short and simple.

RTX Introduction

Many microcontroller applications require simultaneous execution of multiple jobs or tasks. For such applications, an RTOS allows flexible scheduling of system resources (CPU, memory, etc.) to several tasks.

With RTX, you write and compile programs using standard C. Only a few deviations from standard C are required in order to specify the task ID and priority. RTX-166 programs require the inclusion of the RTX166.H or RTX166T.H header file also. RTX_CONFIG.C is required on ARM devices. By selecting the operating system through the dialog **Options for Target – Target**, the linker, L166, included in µVision, links the appropriate RTX-166 library file.

Single Task Program

A standard C program starts execution with the *main* function. In an embedded application, the *main* function is usually coded as an endless loop and can be thought of as a single task that is executed continuously. For example:

```
int counter;

main (void)  {
  counter = 0;

  while (1)  {                              // repeat forever
    counter++;                              // increment counter
  }
}
```

Round-Robin Task Switching

Round-Robin task switching allows a quasi-parallel, simultaneous execution of several tasks. Each task is executed for a predefined period. A timeout suspends the execution of a task and causes another task to be started. The following example uses this round-robin task switching technique.

Program execution starts with *job0*, as an RTOS task function. The RTX function *os_tsk_create* marks *job1* as ready for execution. The task functions *job0* and *job1* are simple counting loops. After its time slot has been consumed, RTX suspends the execution of *job0* and begins execution of *job1*. As soon as its time slot is consumed, the system continues with *job0*.

Simple RTX Program using Round-Robin Task Switching

```
int counter0;
int counter1;

__task1 void job0 (void)  {
  os_tsk_create (job1, 1);                      // start job 1

  while (1)  {                                  // endless loop
    counter0++;                                 // Increment counter 0
  }
}

__task void job1 (void)  {
  while (1)  {                                  // Endless loop
    counter1++;                                 // Increment counter 1
  }
}

main (void)  {                                  // the main function
  os_sys_init (job0);                           // starts only job 0
}
```

The Wait Functions

The RTX kernels provide *wait[2]* functions that suspend the execution of the
current task function and wait for the specified event. During that time, a task
waits for an event, while the CPU can execute other task functions.

Wait for Time Delay

RTX uses a hardware timer of the microcontroller device to generate periodic
interrupts (timer ticks). The simplest event is a time delay through which the
currently executing task is interrupted for a specified number of timer ticks.

This following program is similar to the previous example with the exception that
job0 is suspended with *os_dly_wait* after *counter0* has been incremented.
RTX waits three timer ticks until *job0* is ready for execution again. During this
time, *job1* is executed. This function also calls *os_dly_wait* with 5 ticks time
delay. The result: *counter0* is incremented every three ticks and *counter1* is
incremented every five timer ticks.

---

[1] *For **non-ARM** devices the syntax is: void job0 (void) __task   {…}.*

[2] *Within **RTX Tiny** time delays are created with the function os_wait (K_TMO, …).*

Program with Wait for Time Delay

```
int counter0;
int counter1;

__task void job0 (void)  {
  os_tsk_create (job1, 1);                // start job 1

  while (1)  {
    counter0++;                           // Increment counter 0
    os_dly_wait (3);                      // Wait 3 timer ticks
  }
}

__task void job1 (void)  {
  while (1)  {
    counter1++;                           // Increment counter 1
    os_dly_wait (5);                      // Wait 5 timer ticks
  }
}
```

Wait for Event Flags

Event flags are used for task coordination, that is, a task waiting for another task
to raise an event flag. If an event flag was set previously, the task continues its
execution.

Program with Wait for Event Flag
(*job1* waits for *job0* and counts overflow of *i0*)

```
long i0, save_i0, i1;
OS_TID id1;                              // task ID for event transmits

__task void job0 (void)  {
  id1 = os_tsk_create (job1, 1);         // start job 1

  while (1)  {
    i0++;
    if (i0 > 1000000)  {                 // when i1 reaches 1000000
      i0 = 0;                            // clear i1
      os_evt_set (1, id1);               // set event '1' on job1
    }
  }
}

__task void job1 (void)  {
  while (1)  {
    os_evt_wait_or (1, 0xffff);          // wait for event '1'
    save_i0 = i0;                        // save value of i0
    i1;                                  // count events in i1
  }
}
```

Preemptive Task Switching

Tasks with the same priority[1] (example above) need a round-robin timeout or an explicit call to a RTX *wait* function to execute other tasks. Therefore, in the example above, the value of *save_i0* is not zero, as you might have expected. If *job1* has a higher task priority than *job0*, execution of *job1* starts instantly and the value of *save_i0* will be zero. *job1* preempts execution of *job0* (this is a very fast task switch requiring a few ms only).

Start *job1* with Higher Task Priority

```
   :
__task void job0 (void)  {
  id1 = os_tsk_create (job1, 2);          // start job 1 with priority 2
   :
   :
}
```

---

[1] **RTX Tiny** *does not offer task priorities. Instead,* **RTX Tiny** *has one event flag per task, called signal, and uses the function os_wait (K_SIG, …) to wait for this signal flag.*

Mailbox Communication

A mailbox is a FIFO (first in – first out) buffer for transferring messages between task functions. Mailbox functions accept pointer values, typically referencing memory buffers. However, by using appropriate type casts, you may pass any integer 32-bit data type.

Program with Mailbox Communication[1]

```
os_mbx_declare(v_mail, 20);               // mailbox with 20 entries

__task void job0 (void)  {
  int i, res;

  os_mbx_init (v_mail, sizeof (v_mail));   // create mailbox first
  os_tsk_create (job1, 2);                 // before waiting tasks

  for (i = 0; i < 30; )  {                         // send 30 mail
    res = os_mbx_send (v_mail, (void *) i, 1000);
    if (res == OS_R_OK)  i++;              // check that mail send OK
  }
  os_tsk_delete_self ();                   // when done delete own task
}

__task void job1 (void)  {
  int v, res;

  while (1)  {
    res = os_mbx_wait (v_mail, (void **) &v, 0xFFFF);  // receive mail
    if (res == OS_R_OK || s == OS_R_MBX)  {            // check status
      printf ("\nReceived v=%d res=%d", v, res);       // use when correct
    }
  }
}
```

The task *job0* uses a mailbox to send information to *job1*. When *job1* runs with a higher priority than *job0*, the mail is instantly delivered. The mailbox buffers up to 20 messages when *job1* runs with the same or lower priority than *job0*.

The *os_mbx_send* and *os_mbx_wait* functions provide a timeout value that allows function termination when no mail can be delivered within the timeout period.

---

[1] *When creating high-priority tasks using a mailbox, initialize the mailbox before it might be used by a high-priority task.*

Semaphores

Semaphores are utilized to synchronize tasks within an application. Although they have a simple set of calls to the operating system, they are the classic solution in preventing race conditions. However, they do not resolve resource deadlocks. RTX ensures that atomic operations used with semaphores are not interrupted.

Binary Semaphores

Synchronizing two tasks is the simplest use case of a semaphore:

```
os_sem semA;                                    // declare the semaphore

__task void job0 (void)  {
  os_sem_init(semA, 0);

  while(1) {
     do_func_A();
     os_sem_send(semA);                         // free the semaphore
  }
}

__task void job1 (void)  {

  while(1) {
     os_sem_wait(semA, 0xFFFF);                 // wait for the semaphore
     do_func_B();
  }
}
```

In this case the semaphore is used to ensure the execution of *do_func_A()* prior to executing *do_func_B()*.

Counting Semaphores (Multiplex)

Use a multiplex to limit the number of tasks that can access a critical section of code. For example, a routine to access memory resources and that supports a limited number of calls only.

```
os_sem mplxSema;                          // declare the semaphore

__task void job0 (void)  {
  os_sem_init (mplxSema, 5);              // init semaphore with 5 tokens

  while(1) {
     os_sem_wait (mplxSema);             // acquire a token
     processBuffer();
     os_sem_send (mplxSema);             // free the token
  }
}
```

In this example, we initialize the multiplex semaphore with five tokens. Before a task can call *processBuffer()*, it must acquire a semaphore token. Once the function has completed, it returns the token to the semaphore. If more than five calls attempt to invoke *processBuffer()*, the sixth must wait until one of the five running tasks returns its token. Thus, the multiplex semaphore ensures that a maximum of five calls can use *processBuffer()* simultaneously.

Interrupt Service Routines

An interrupt is an asynchronous signal from the hardware or software that forces the microcontroller to save the execution state. Interrupts trigger a context switch to an interrupt handler. Software interrupts are implemented as instructions in the instruction set of the microcontroller and work similar to hardware interrupts. Interrupts can be classified as a:

- Maskable interrupt (IRQ) – a hardware interrupt that can be ignored by setting a bit in a bit-mask

- Non-maskable interrupt (NMI) – a hardware interrupt that cannot be configured and thus cannot be ignored

- Software interrupt – generated within a processor by executing an instruction

RTX ensures that interrupts execute correctly and leaves the machine in a well-defined state. Interrupt service routines, also known as interrupt handlers, are used to service hardware devices and transitions between operation modes, such as system calls, system timers, disk I/O, power signals, keystrokes, watchdogs; other interrupts transfer data using UART or Ethernet.

Hints for working with interrupt functions in RTX:

- Functions that begin with *os_* can be called from a task but not from an interrupt service routine

- Functions that begin with *isr_* can be called from an **IRQ** interrupt service routine but not from a task. Never use them from **FIQ**.

- Never enable any **IRQ** interrupt that calls *isr_* functions before the kernel has been started

- Avoid nesting **IRQ** functions on ARM7/ARM9 targets

- Use short **IRQ** functions to send signals and messages to RTOS tasks

- Interrupt functions are added to applications the same way as in non-RTX projects

- By default, interrupts are globally enabled at startup

Another important concept is the interrupt latency, which is defined as the period between the generation and servicing of that interrupt. This is especially important in systems that need to control machinery in real time, and therefore require low interrupt latency. RTX ensures that a subroutine will finish its execution in an agreed maximum length of time and that the interrupt latency does not exceed a predefined maximum length of time.

The general logic of an ISR looks like the following code example. The interrupt function *ext0_int* sends an event to *process_task* and exits. The task *process_task* processes the external interrupt event. In this example, *process_task* is simple and only counts the number of interrupt events.

```
#define    EVT_KEY 0x00001

OS_TID    pr_task;
int       num_ints;

__irq  void ext0_int (void) {          // external interrupt routine

   isr_evt_set (EVT_KEY, pr_task);     // send event to 'process_task'
   acknYourInterrupt ();               // acknowledge interrupt;
}


__task  void process_task (void)  {
   num_ints =0;

   while(1)  {
      os_evt_wait_or (EVT_KEY, 0xFFFF);
      num_ints++;
   }
}


__task  void init_task (void)  {

   enableYourInterrupt ();
   pr_task = os_tsk_create (process_task, 100);   // create task with prio
   os_tsk_delete_self ();
}
```

Press **F1** to browse through the numerous examples and additional information in the on-line help.

Memory and Memory Pools

The compilers delivered with the Keil development tools provide access to all memory areas, regardless of the microcontroller architecture. Variables can be explicitly assigned to a specific memory space by including a memory type in the declaration, or implicitly assigned based on the memory model. Function arguments and atomic variables that cannot be located in registers are also stored in the default memory area. Accessing the internal data memory is considerably faster than accessing the external data memory. If possible, place often-used variables into the internal memory space and less-used variables into the external memory space.

RTX provides thread-safe and fully reentrant[1] allocation functions for fixed sized memory pools. These functions have a deterministic execution time that is

---

[1] *Variable length memory allocation functions are not reentrant! Disable/enable system timer interrupts using tsk_lock() and tsk_unlock() during the execution of malloc() and free().*

independent of the pool usage. Built-in memory allocation routines enable you to dynamically use the system memory by creating memory pools and use fixed sized blocks from the memory pool. The memory pool needs to be properly initialized to the size of the object.

```c
#include <rtl.h>

os_mbx_declare (MsgBox, 16);                      // declare an RTX mailbox

U32 mpool [16*( 2 * sizeof (U32) ) /4 + 3]; // memory for 16 messages

__task  void rec_task (void);                     // task to receive a message

__task  void send_task (void)  {                  // Task to send a message
  U32 *mptr;

  os_tsk_create (rec_task, 0);
  os_mbx_init (MsgBox, sizeof (MsgBox));     // init mailbox

  mptr = _alloc_box (mpool);                      // alloc. memory for the message

  mptr[0] = 0x3215fedc;                       // set message content
  mptr[1] = 0x00000015;

  os_mbx_send (MsgBox, mptr, 0xffff);    // Send the message to 'MsgBox'

  os_tsk_delete_self ();
}

__task  void rec_task (void)  {
  U32 *rptr, rec_val[2];

  os_mbx_wait (MsgBox, &rptr, 0xffff);    // Wait for message
  rec_val[0] = rptr[0];                       // Store content to 'rec_val'
  rec_val[1] = rptr[1];

  _free_box (mpool, rptr);                     // Release the memory block

  os_tsk_delete_self ();
}

void main (void)  {

  _init_box (mpool, sizeof (mpool), sizeof (U32));
  os_sys_init (send_task);
}
```

To send a message object of a variable size and use the variable size memory block, you must use the memory allocation functions, which can be found in **stdlib.h**.

RTX and ARTX166 Function Overview

Function Group	RTX	ARTX166
Task Management	create-task, delete-task, pass-task, change-priority, running-task-id, running-task-priority, lock-task, unlock-task, system-init, system-priority	create-task, delete-task, pass-task, change-priority, running-task-id, running-task-priority, lock-task, unlock-task, system-init, define-task
Event/Signal Functions	clear-event, get-event, set-event, wait-event, isr-set-event	clear-event, get-event, set-event, wait-event, isr-set-event
Semaphore Functions	initialize -semaphore, send-semaphore, wait-semaphore, isr-send-semaphore	initialize -semaphore, send-semaphore, wait-semaphore, isr-send-semaphore
Mailbox Functions	check-mbx, declare-mbx, initialize -mbx, send-mbx, wait-mbx, isr-receive-mbx, isr-send-mbx, isr-check-mbx	check-mbx, declare-mbx, initialize -mbx, send-mbx, wait-mbx, isr-receive-mbx, isr-send-mbx,
Memory Management	create-pool, check-pool, get-block, free-block	
Mutex Management	initialize-mutex, release-mutex, wait-mutex	initialize-mutex, release-mutex, wait-mutex
System Clock (Timer-Ticks)	delay-task, wake-up-task, set-slice, create-timer, kill-timer, call-timer	delay-task, wake-up-task, set-slice, create-timer, kill-timer, call-timer
Generic WAIT Function	interval-wait	interval-wait

RTX and ARTX166 Technical Data

Technical Data	RTX		ARTX166
max Tasks	250		250
Events/Signals	16 per task		16 per task
Semaphores, Mailboxes, Mutexes	unlimited		unlimited
min RAM	2 – 3 KBytes		500 Bytes
	ARM7/ARM9	Cortex-Mx	
max Code Space	4.2 KBytes	4.0 KBytes	4.0 KBytes
Hardware Needs	1 on-chip timer	SysTick timer	1 on-chip timer
Task Priorities	1 – 254	1 – 254	1-127
Context Switch	< 7 µsec @ 60 MHz	< 4 µsec @ 72 MHz	< 15 µsec @ 20 MHz
Interrupt Lockout	3.1 µsec @ 60 MHz	not disabled by RTX	0.2 µsec @ 20 MHz

RTX51 Tiny and RTX166 Tiny Function Overview

Function Group	RTX51 Tiny	RTX166 Tiny
Task Management	create-task, delete-task, running-task-id, switch-task, set-ready, isr-set-ready	create-task, delete-task, running-task-id, delay-task
Signal Functions	send-signal, clear-signal, isr-send-signal	send-signal, clear-signal, isr-send-signal, wait-signal
System Clock (Timer-Ticks)	reset-interval	delay-task
Generic WAIT Function	wait	wait

RTX51 Tiny and RTX166 Tiny Technical Data

Technical Data	RTX51 Tiny	RTX166 Tiny
max Tasks	16	32
Signals	16	32 max
RAM	7 + 3 Bytes/Task	8 + 4 Bytes/Task
max Code Space	900 Bytes	1.5 KBytes
Hardware Needs	No Timer	1 on-chip Timer
Context Switch	100-700 Cycles	400 – 4000 Cycles
Interrupt Lockout	< 20 Cycles	< 4 µsec, 0 ws.

Chapter 5. Using µVision

The µVision IDE is, for most developers, the easiest way to create embedded system programs. This chapter describes commonly used µVision features and explains how to use them.

General Remarks and Concepts
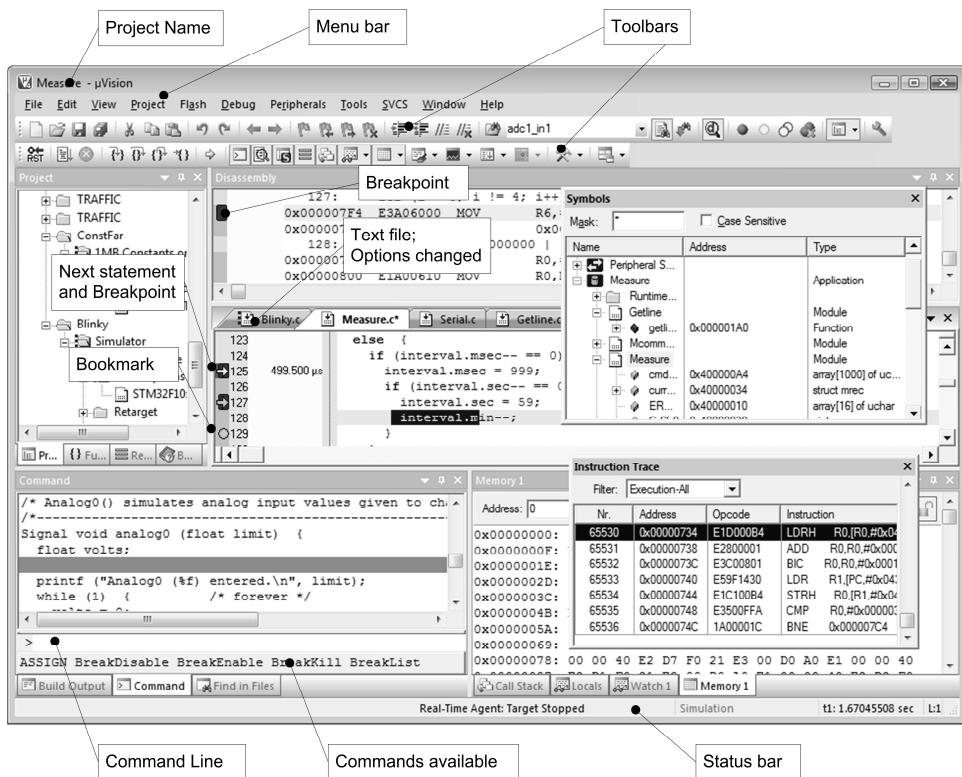
Before we start to describe how to use µVision, some general remarks, common to many screens[1] and to the behavior of the development tool, are presented. In our continuous effort to deliver best-in-class development tools, supporting you in your daily work, µVision has been built to resemble the look-and-feel of widespread applications. This approach decreases your learning curve, such that you may start to work with µVision right away.

Based on the concept of windows:

- µVision windows can be re-arranged, tiled, and attached to other screen areas or windows respectively

- It is possible to drag and drop windows, objects, and variables

- A Context Menu, invoked through the right mouse button, is provided for most objects

- You can use keyboard shortcuts and define your own shortcuts

- You can use the abundant features of a modern editor

- Menu items and Toolbar buttons are grayed out when not available in the current context

- Graphical symbols are used to resemble options, to mark unsaved changes, or reveal objects not included into the project

- Status Bars display context-driven information

- You can associate µVision to third-party tools

---

[1]*The screenshots presented in the next chapters have been taken from different example programs and several microcontroller architectures to resemble the main feature, sometimes the special feature, of that topic. The same window, dialog, or tab category will look slightly different for other microcontroller architectures.*
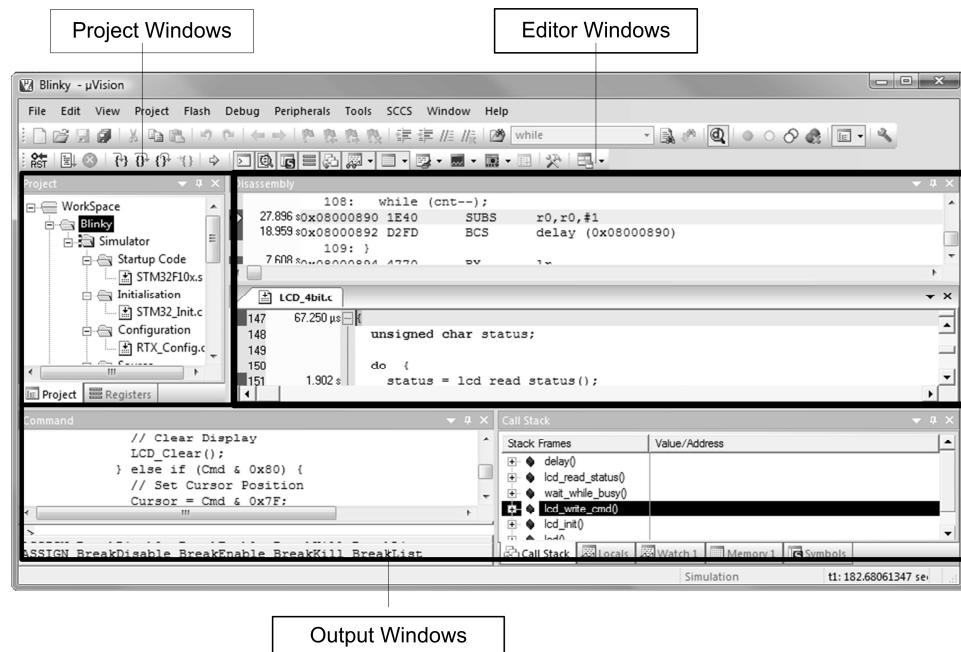
To launch μVision click the μVision icon on your desktop or select μVision from the Start Menu.

Window Layout Concepts

You can set up your working environment[1] in μVision at your discretion. Nevertheless, let us define three major screen areas. The definition will help you to understand future comments, illustrations, and instructions.

---

[1] *Any window can be moved to any other part of the μVision screen, or even outside of μVision to any other physical screen, with the exception of the objects related to the Text Editor.*

The **Project Windows** area is that part of the screen in which, by default, the Project Window, Functions Window, Books Window, and Registers Window are displayed.

Within the **Editor Windows** area, you are able to change the source code, view performance and analysis information, and check the disassembly code.

The **Output Windows** area provides information related to debugging, memory, symbols, call stack, local variables, commands, browse information, and find in files results.

If, for any reason, you do not see a particular window and have tried displaying/hiding it several times, please invoke the default layout of µVision through the **Window – Reset Current Layout** Menu**.**

Positioning Windows
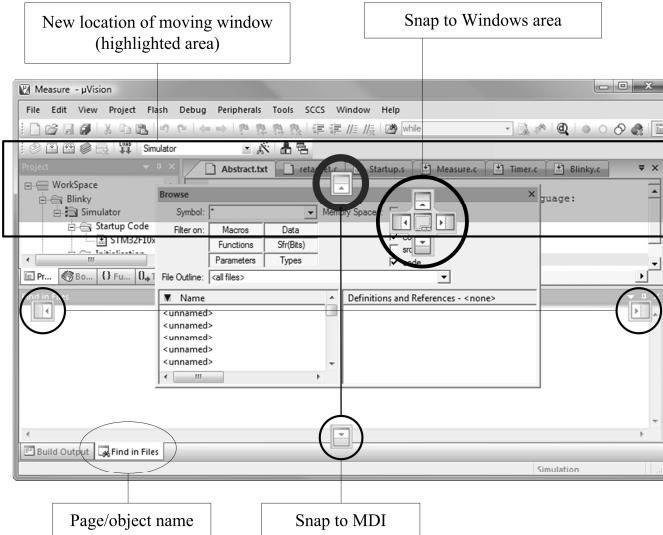
The µVision windows may be placed onto any area of the screen, even outside of the µVision frame, or to another physical screen.

▪ Click and hold the **Title Bar**[1] of a window with the left mouse button

▪ Drag the window to the preferred area, or onto the preferred control, and release the mouse button

Please note, source code files cannot be moved outside of the **Editor Windows**[2].

Invoke the **Context Menu** of the window's **Title Bar** to change the docking attribute of a window object. In some cases, you must perform this action before you can drag and drop the window.

µVision displays docking helper controls[3], emphasizing the area where the window will be attached. The new docking area is represented by the section highlighted in blue. Snap the window to the Multiple Document Interface (MDI) or to a Windows area by moving the mouse over the preferred control.

---

[1] *You may click the page/object name to drag and drop the object.*

[2] *Source code files stay in the Text Editor's window.*

[3] *Controls indicate the area of the new window position. The new position is highlighted.*

µVision Modes

µVision operates in two modes: **Build Mode** and **Debug Mode**. Screen settings, Toolbar settings, and project options are stored in the context of the mode. The **File Toolbar** is enabled in all modes, while the **Debug Toolbar** and **Build Toolbar** display in their respective mode only. Buttons, icons, and menus are enabled if relevant for a specific mode.

The standard working mode is **Build Mode**. In this mode you write your application, configure the project, set preferences, select the target hardware and the device; you will compile, link, and assemble the programs, correct the errors, and set general settings valid for the entire application.

In **Debug Mode**, you can also change some general options and edit source code files, but these changes will only be effective after you have switched back to **Build Mode**, and rebuild your application. Changes to debug settings are effective immediately.

Menus

The **Menu** bar provides access to most µVision commands including file operations, editor operations, project maintenance, development tool settings, program debugging, window selection and manipulation, and on-line help.

File Menu

The **File** Menu includes commands that open, save, print, and close source files. The **Device Database** and **License Manager** dialogs are accessed from this menu.

Edit Menu

The **Edit** Menu includes commands for editing the source code; undo, redo, cut, copy, paste, and indentation, bookmark functions, various find and replace commands, source outlining functions, and advanced editor functions. Editor configuration settings are also accessed from this menu.

View Menu
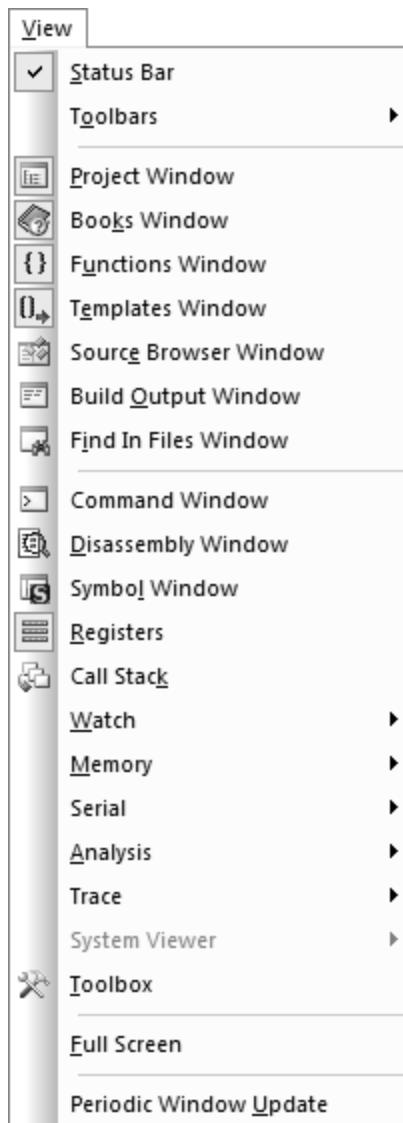
The **View** Menu includes commands to display/hide a variety of windows. You can also enable/disable the **Status Bar**. The **Periodic Window Update** option is useful in **Debug Mode** to force the screens to periodically refresh. If this option has not been selected, you can manually update the screens via the **Toolbox**.

Project Menu

The **Project** Menu includes commands to open, save, and close project files. You can **Export** your project to a previous version of μVision, **Manage** project components, or **Build** the project. In addition, you can set **Options** for the project, group, and file. You can manage multiple projects through the **Multi-Project Workspace…** Menu.
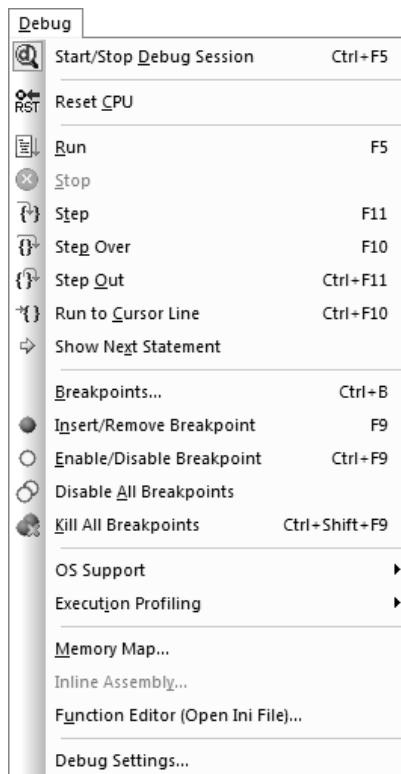
Flash Menu

The **Flash** Menu includes commands you can use to configure, erase, and program Flash memory for your embedded target system.

View
✓ Status Bar
Toolbars ▶
Project Window
Books Window
Functions Window
Templates Window
Source Browser Window
Build Output Window
Find In Files Window
Command Window
Disassembly Window
Symbol Window
Registers
Call Stack
Watch ▶
Memory ▶
Serial ▶
Analysis ▶
Trace ▶
System Viewer ▶
Toolbox
Full Screen
Periodic Window Update

Debug Menu

The **Debug** Menu includes commands that start and stop a debug session, reset the CPU, run and halt the program, and single-step in high-level and assembly code. In addition, commands are available to manage breakpoints, view RTOS Kernel information, and invoke execution profiling. You can modify the memory map and manage debugger functions and settings.

Debug		
Start/Stop Debug Session	Ctrl+F5
Reset CPU	
Run	F5
Stop	
Step	F11
Step Over	F10
Step Out	Ctrl+F11
Run to Cursor Line	Ctrl+F10
Show Next Statement	
Breakpoints...	Ctrl+B
Insert/Remove Breakpoint	F9
Enable/Disable Breakpoint	Ctrl+F9
Disable All Breakpoints	
Kill All Breakpoints	Ctrl+Shift+F9
OS Support	▶
Execution Profiling	▶
Memory Map...	
Inline Assembly...	
Function Editor (Open Ini File)...	
Debug Settings...	

Tools Menu

Configure and run PC-Lint or set up your own tool shortcuts to third party utilities.

SVCS Menu

The **SVCS** Menu allows you to configure and integrate your project development with third-party version control systems.
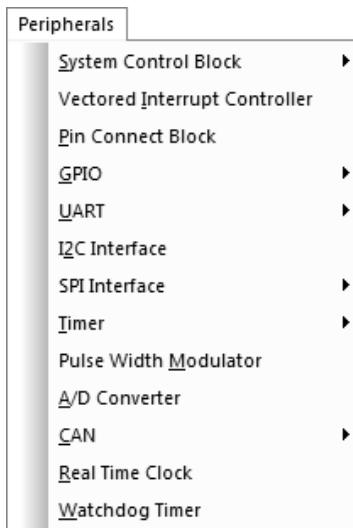
Help Menu

The **Help** Menu includes commands to start the on-line help system, to list information about on-chip peripherals, to access the knowledgebase, to contact the Technical Support team, to check for product updates, and to display product version information.

Help	
µVision Help	
Open Books Window	
Simulated Peripherals for 'LPC2129'	
Internet Support Knowledgebase	
Contact Support	
Check for Update	
About µVision...	
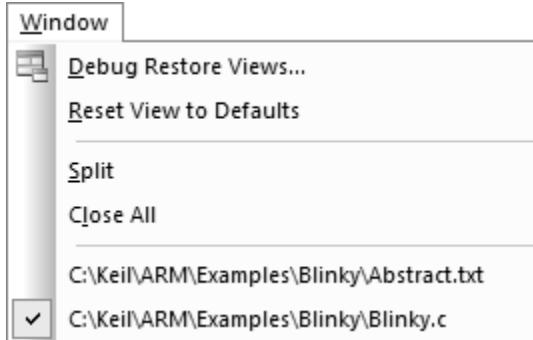
Peripherals Menu

The **Peripherals** Menu includes dialogs to display and change on-chip peripheral settings. The content of this menu is tailored to show the specific microcontroller options selected for your application. Dialogs are typically available for System Configuration, Interrupts, UARTs, I2C, Timer/Counters, General Purpose I/O, CAN, Pulse-Width Modulators, Real-Time Clocks, and Watchdog Timers. This menu is active in **Debug Mode** only.

Peripherals	
System Control Block	▶
Vectored Interrupt Controller	
Pin Connect Block	
GPIO	▶
UART	▶
I2C Interface	
SPI Interface	▶
Timer	▶
Pulse Width Modulator	
A/D Converter	
CAN	▶
Real Time Clock	
Watchdog Timer	

Window Menu

The **Window** Menu includes commands to split, select, and close various windows in the **Text Editor**.

In addition, you can define your own screen layouts through the **Debug Restore Views…** dialog, and switch back and forth between the screen layouts you defined.

Window
Debug Restore Views...
Reset View to Defaults
Split
Close All
C:\Keil\ARM\Examples\Blinky\Abstract.txt
✓ C:\Keil\ARM\Examples\Blinky\Blinky.c

Restore the default layout through **Reset View to Defaults** at any time. Currently open source code windows are listed at the bottom of the **Window** Menu.

Toolbars and Toolbar Icons

The µVision IDE incorporates several Toolbars with buttons for the most commonly used commands.

- The **File Toolbar** contains buttons for commands used to edit source files, to configure µVision, and to set the project specific options
- The **Build Toolbar** contains buttons for commands used to build the project
- The **Debug Toolbar** contains buttons for commands used in the debugger

The **File Toolbar** is always available, while the **Build Toolbar** and **Debug Toolbar** will display in their context. In both modes, **Build Mode** and **Debug Mode**, you have the option to display or hide the applicable Toolbars.

File Toolbar

	New File – opens an empty text window
	Open File – dialog to open an existing file
	Save File – saves the contents of the current file
	Save All – saves changes in all open files
	Cut – deletes the selected text and copies it to the clipboard
	Copy – copies the currently selected text to the clipboard
	Paste – inserts text from the clipboard to the current cursor position
	Undo changes – removes prior changes in an edit window
	Redo changes – restores the last change that was undone
	Navigate Backwards – moves cursor to its former backward position

➡   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

RST	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
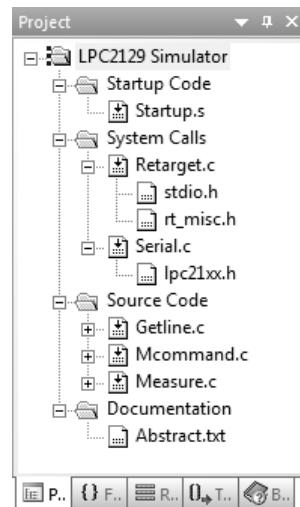
Print – opens the printer dialog

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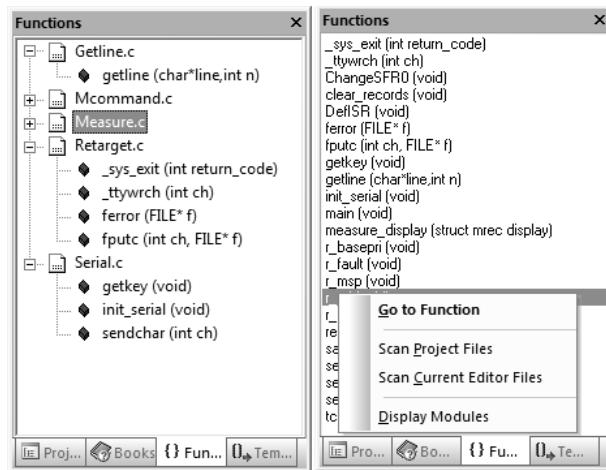
Project Windows display information about the
current project. The tabs at the bottom of this area
provide access to:

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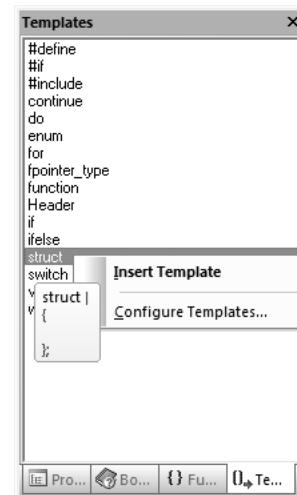
Double-click a function to jump to its definition. Invoke its **Context Menu** to toggle the displaying mode of this window or scan the files.

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:
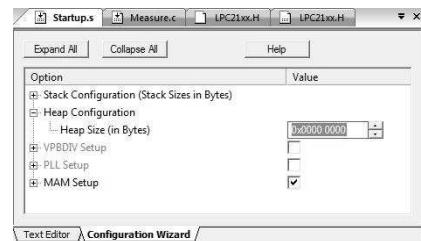
- Write, edit, and debug source files. Press **F1** on language elements for help.

- 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**.
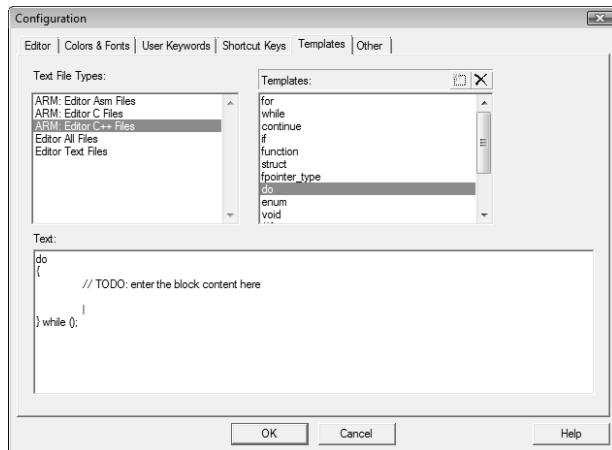
Editor Configuration

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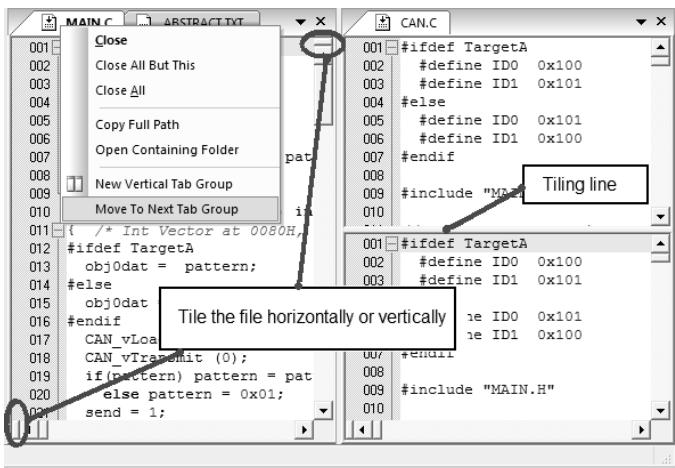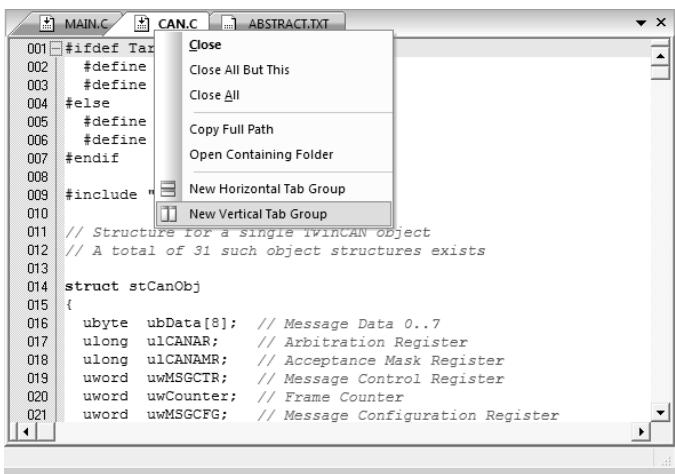
You can view text files in the Editor side by side. Invoke the **Context Menu** of the file tab and choose a horizontal or vertical arrangement.

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.*
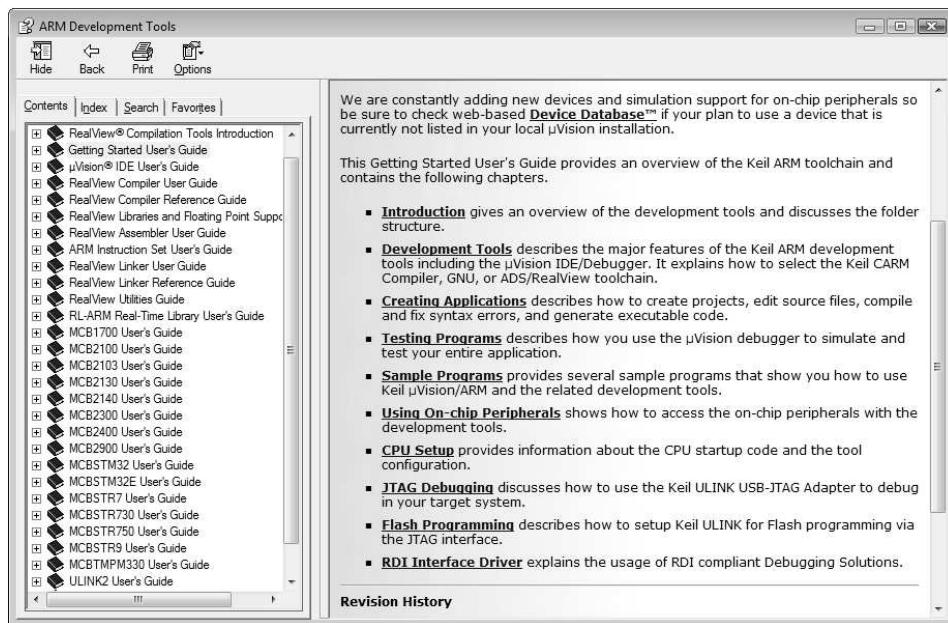
Other Windows and Dialogs

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
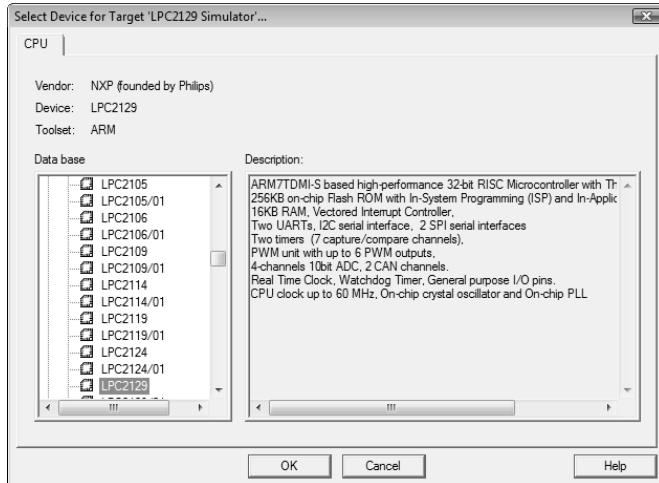
Selecting the Folder and Project Name

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.

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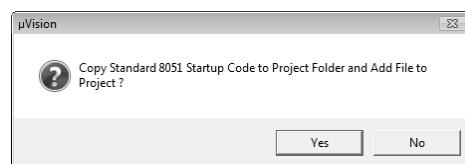
[1] *For some devices, µVision requires additional parameters you must enter manually. Please read the device description in the Select Device dialog carefully, as it may contain extra instructions for the device configuration.*

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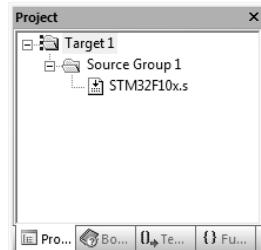
All Keil tools include chip-specific startup code for most of the devices listed in the **Device Database**. Copy the startup code to your project folder and modify it there only. μVision automatically displays a dialog to copy the startup code into your project folder. Answer this question with **YES**.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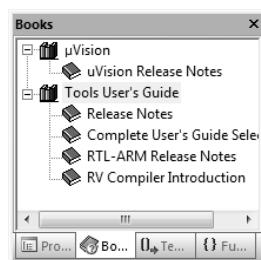
Using the Project Windows

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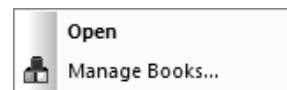
The **Books Window**, also part of the **Project Windows**, provides the Keil product manuals, data sheets, and programmer's guides for the selected microcontroller. Double-click a book to open it.

---

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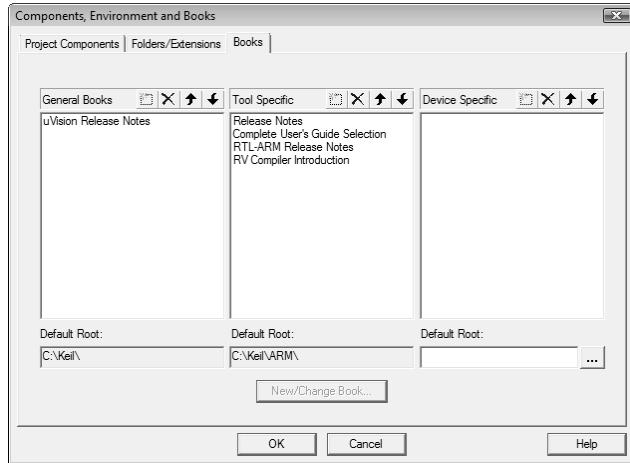
[2] *Library and add-on projects need no startup code.*

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

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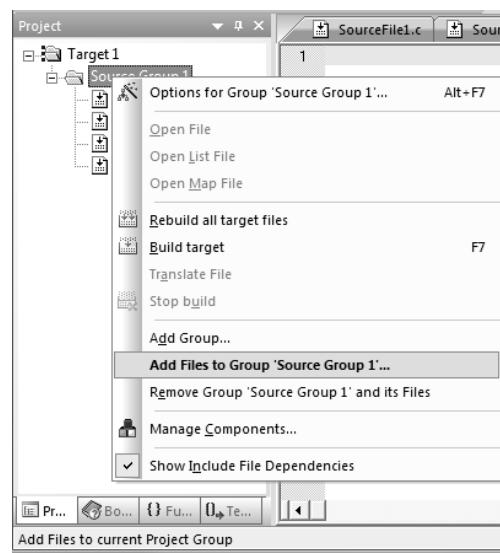
[1] *Most microcontroller manuals are part of the toolset, or are available on the Keil Development Tools CD-ROM.*

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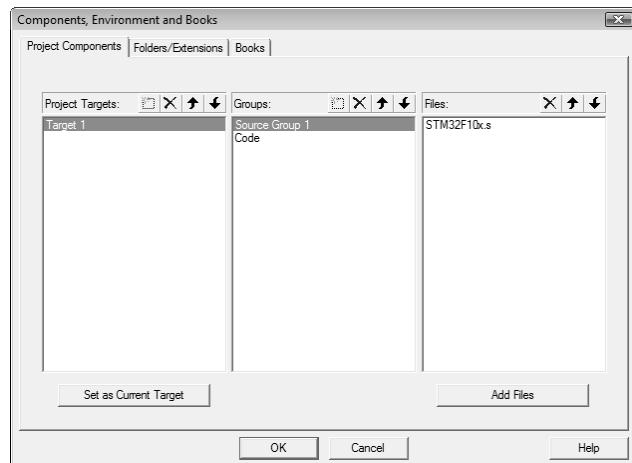
Use the **Components, Environment, and Books…** dialog to manage
your Targets, Groups, and Files configuration

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.
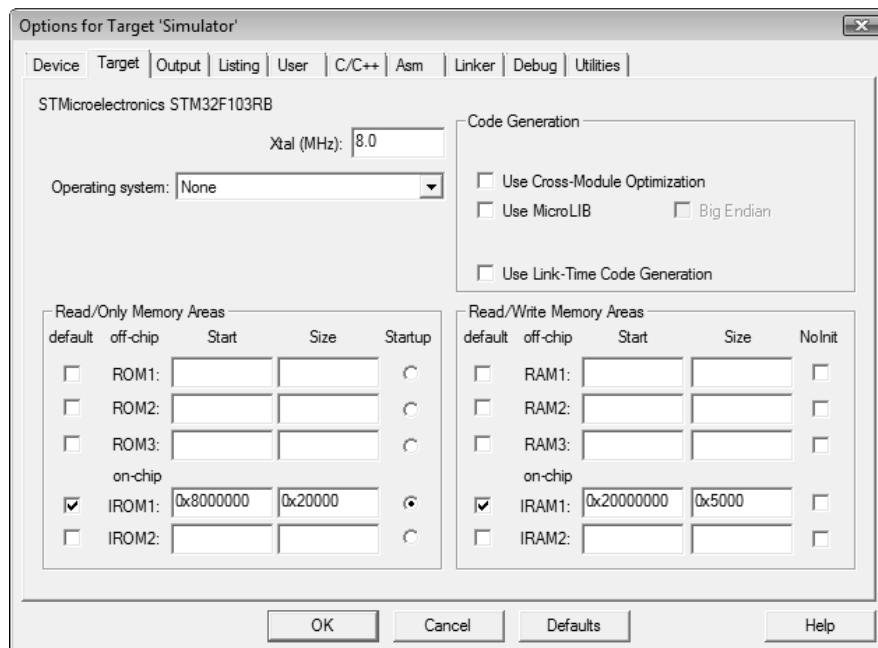

Insert - create a new target or group

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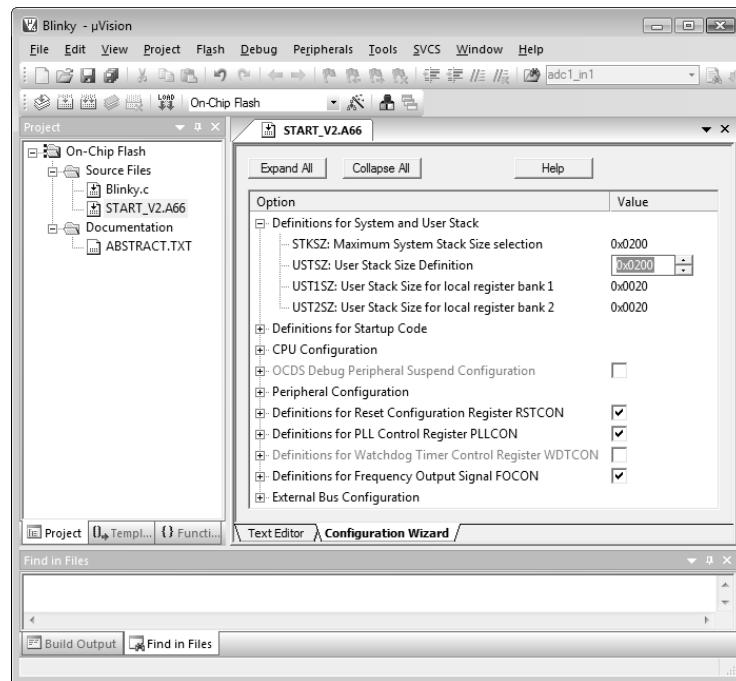
Treat **Target** options similar to general options. They are valid for the entire project and for that target. Some options can be defined at the group level and on individual files. File-level options will supersede group-level options, which in turn, supersede the options set at the target level.

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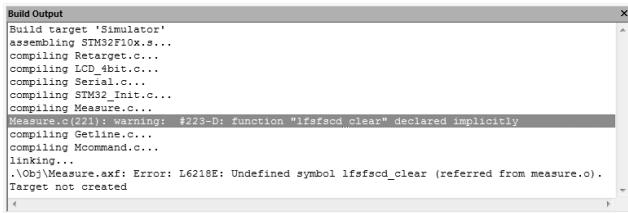
Several commands are available from the **Build Toolbar** or **Project** Menu to assemble, compile, and link the files of your project. Before any of these actions are executed, files are saved.

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.
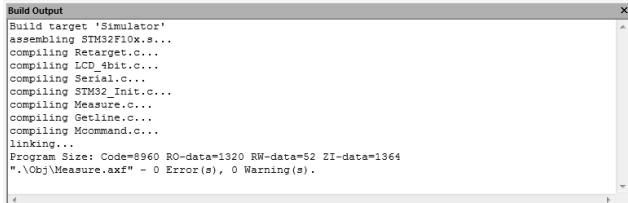
```
Build Output                                                                     ×
Build target 'Simulator'
assembling STM32F10x.s...
compiling Retarget.c...
compiling LCD_4bit.c...
compiling Serial.c...
compiling STM32_Init.c...
compiling Measure.c...
Measure.c(221): warning:  #223-D: function "lfsfscd_clear" declared implicitly
compiling Getline.c...
compiling Mcommand.c...
linking...
.\Obj\Measure.axf: Error: L6218E: Undefined symbol lfsfscd_clear (referred from measure.o).
Target not created
```

µVision displays the message **0 Error(s), 0 Warning(s)** on successful completion of the build process. Though existing warnings do not prevent

```
Build Output                                                                     ×
Build target 'Simulator'
assembling STM32F10x.s...
compiling Retarget.c...
compiling LCD_4bit.c...
compiling Serial.c...
compiling STM32_Init.c...
compiling Measure.c...
compiling Getline.c...
compiling Mcommand.c...
linking...
Program Size: Code=8960 RO-data=1320 RW-data=52 ZI-data=1364
".\Obj\Measure.axf" - 0 Error(s), 0 Warning(s).
```

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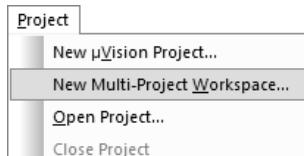
Sometimes, application development requires working with more than one project at the same time. With single projects, that requires closing the current project and opening the new project. The µVision Multi-Project feature allows you to define a group of projects as a Multi-Project file and to work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

Project
New µ<u>V</u>ision Project...
New Multi-Project <u>W</u>orkspace...
<u>O</u>pen Project...
<u>C</u>lose Project

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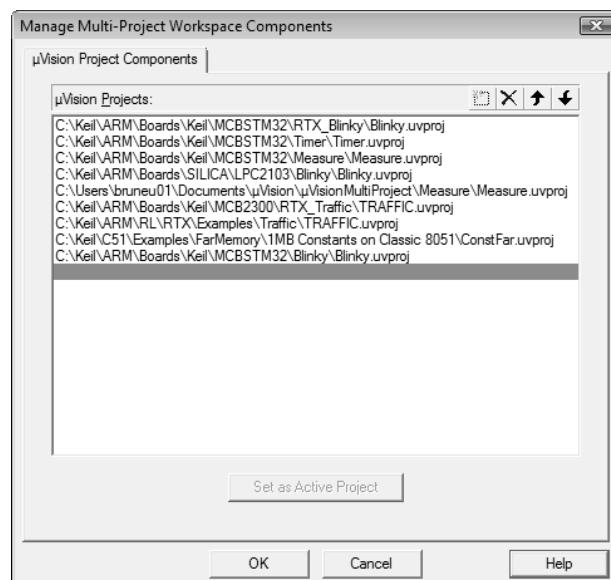
Invoke the **Manage Multi-Project Workspace Components** dialog through the **Project – Manage – Multi Project Workspace…** Menu, or
use the **Manage Multi-Project Workspace…** button of the **Build Toolbar**.

　　　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.
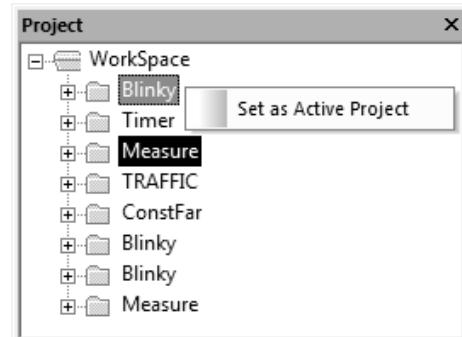
Activating a Multi-Project

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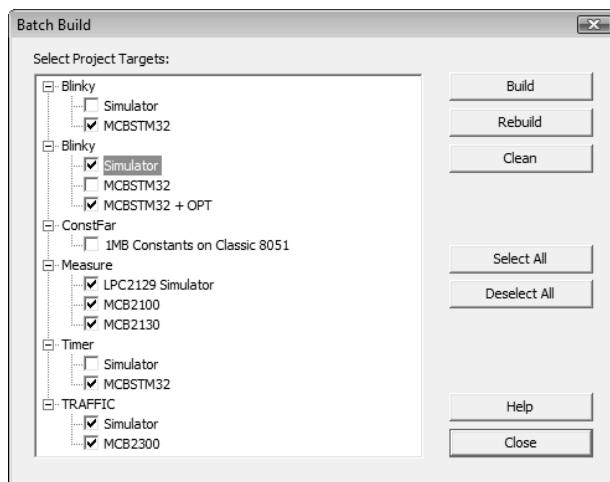
Use the **Batch Build**[1] command from the **Build Toolbar** or from the **Project – Batch Build** Menu to build, re-build, or clean the Project Targets.

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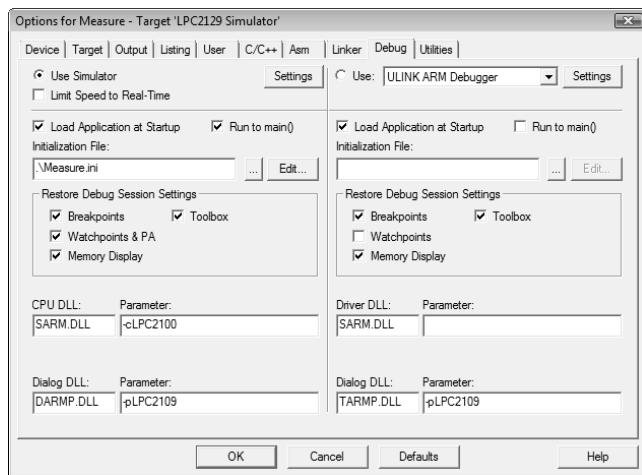
[2] *Programs run on your target hardware. You can debug your application with restrictions.*

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:
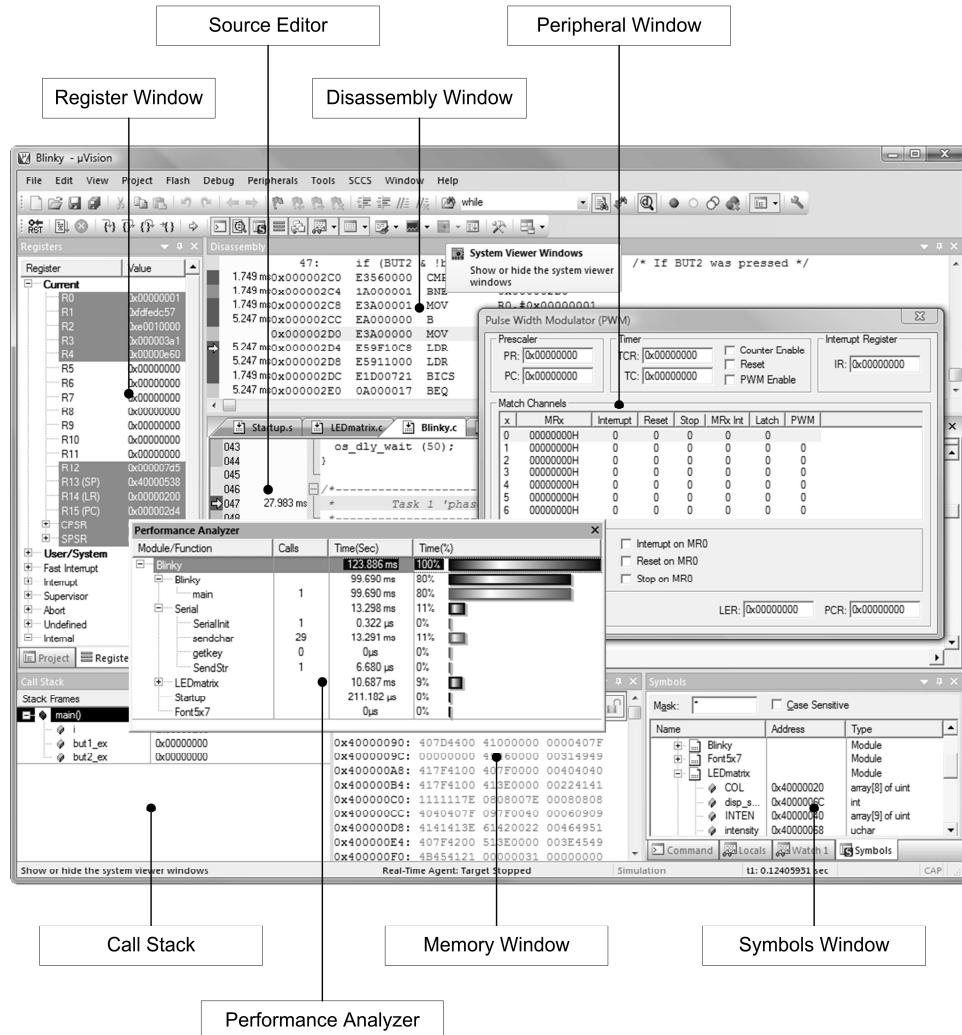
- Runs programs using the ARM7, ARM9, Thumb, Thumb2, 8051, C166/XE166/XC2000 instruction sets

- 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**.

Source Editor

Peripheral Window

Register Window

Disassembly Window

Call Stack

Memory Window

Symbols Window

Performance Analyzer

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
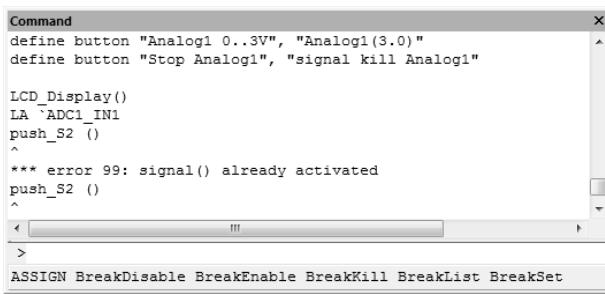
Besides the disabled build commands, you may not:

- 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                                              ×
define button "Analog1 0..3V", "Analog1(3.0)"
define button "Stop Analog1", "signal kill Analog1"

LCD_Display()
LA `ADC1_IN1
push_S2 ()
^
*** error 99: signal() already activated
push_S2 ()
^

 >
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet
```
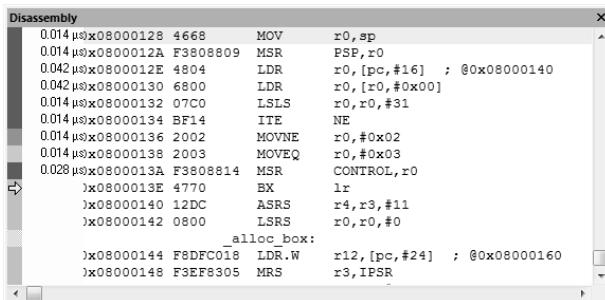
Command Window. Valid instructions will rise on its status bar with hints to parameters and parameter options. Insert expressions to view or modify the content of registers, variables, and memory areas. You can invoke debugger script functions as well. We strongly advise you to make use of the detailed 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.

```
Disassembly                                               ×
0.014 µs)x08000128 4668      MOV      r0,sp
0.014 µs)x0800012A F3808809  MSR      PSP,r0
0.042 µs)x0800012E 4804      LDR      r0,[pc,#16]  ; @0x08000140
0.042 µs)x08000130 6800      LDR      r0,[r0,#0x00]
0.014 µs)x08000132 07C0      LSLS     r0,r0,#31
0.014 µs)x08000134 BF14      ITE      NE
0.014 µs)x08000136 2002      MOVNE    r0,#0x02
0.014 µs)x08000138 2003      MOVEQ    r0,#0x03
0.028 µs)x0800013A F3808814  MSR      CONTROL,r0
⇨      )x0800013E 4770       BX       lr
       )x08000140 12DC       ASRS     r4,r3,#11
       )x08000142 0800       LSRS     r0,r0,#0
                    _alloc_box:
       )x08000144 F8DFC018   LDR.W    r12,[pc,#24]  ; @0x08000160
       )x08000148 F3EF8305   MRS      r3,IPSR
```

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.

Registers	
Register	Value
Core	
R0	0x00000004
R1	0x20000008
R2	0x00000100
R3	0x080004f1
R4	0x08000520
R5	0x08000520
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000208
R14 (LR)	0x08000391
R15 (PC)	0x08000488
xPSR	0x01000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	27936794
Sec	0.38801158

Project   Registers

Memory Window

Monitor memory areas through four distinct **Memory Windows**.
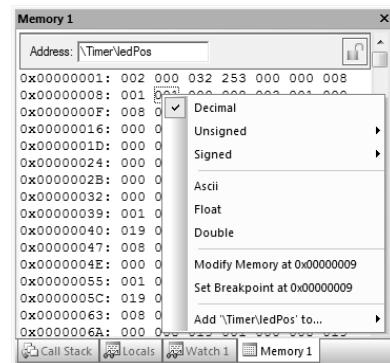
🔳   Open the **Memory Window** from the **Debug Toolbar** or the
View – Memory – Memory[x**]** Menu

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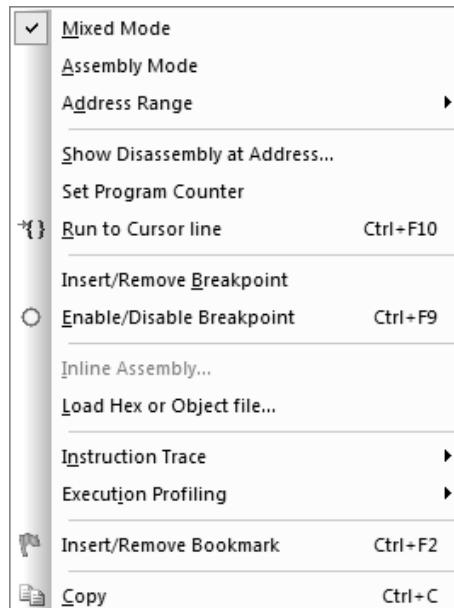
You can double-click the gray sidebar of the **Editor Window** or **Disassembly Window** to set a breakpoint, or use the breakpoint buttons of the **File Toolbar**.

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.

✓	Mixed Mode	
	Assembly Mode	
	Address Range	▶
	Show Disassembly at Address...	
	Set Program Counter	
⁀{}	Run to Cursor line	Ctrl+F10
	Insert/Remove Breakpoint	
○	Enable/Disable Breakpoint	Ctrl+F9
	Inline Assembly...	
	Load Hex or Object file...	
	Instruction Trace	▶
	Execution Profiling	▶
⚑	Insert/Remove Bookmark	Ctrl+F2
▤	Copy	Ctrl+C

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.
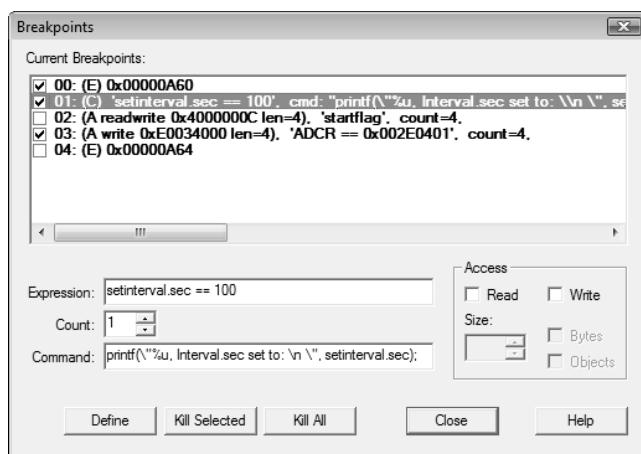
Breakpoints Window

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**.
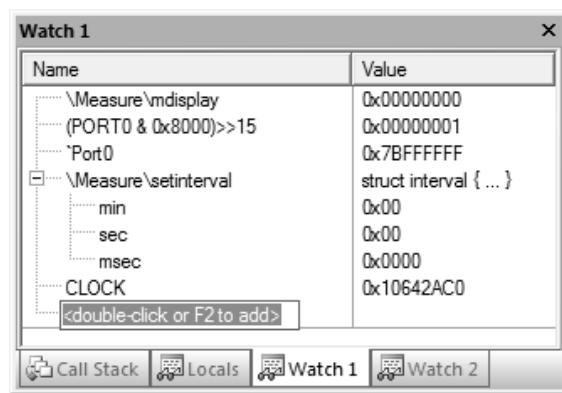
Watchpoints and Watch Window

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.
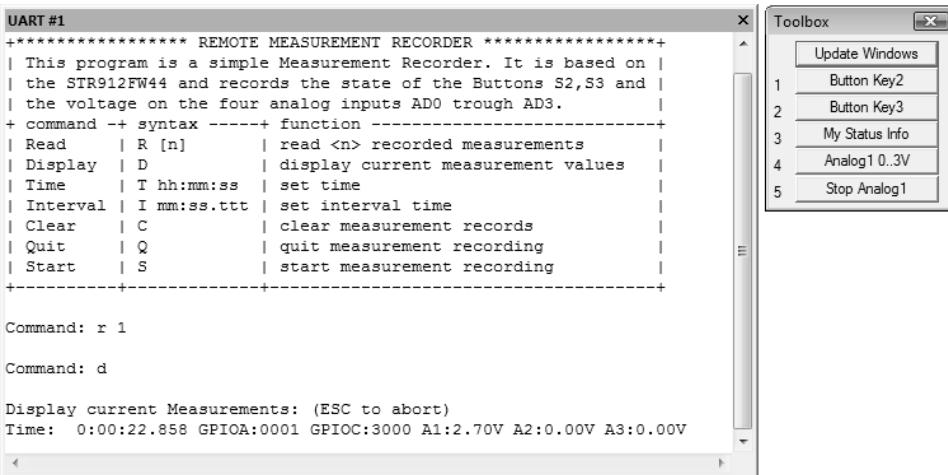
Watchpoint Commands

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.

```
UART #1                                                                    ×    Toolbox              
+**************** REMOTE MEASUREMENT RECORDER ****************+   ▲       Update Windows  
| This program is a simple Measurement Recorder. It is based on |      1   Button Key2     
| the STR912FW44 and records the state of the Buttons S2,S3 and |      2   Button Key3     
| the voltage on the four analog inputs AD0 trough AD3.         |      3   My Status Info  
+ command -+ syntax -----+ function ---------------------------+      4   Analog1 0..3V   
| Read     | R [n]       | read <n> recorded measurements       |      5   Stop Analog1    
| Display  | D           | display current measurement values   |                         
| Time     | T hh:mm:ss  | set time                             |                         
| Interval | I mm:ss.ttt | set interval time                    |                         
| Clear    | C           | clear measurement records            |  ▤                      
| Quit     | Q           | quit measurement recording           |                         
| Start    | S           | start measurement recording          |                         
+----------+-------------+--------------------------------------+                         

Command: r 1

Command: d

Display current Measurements: (ESC to abort)
Time:  0:00:22.858 GPIOA:0001 GPIOC:3000 A1:2.70V A2:0.00V A3:0.00V  ▼
◄                                                                ►
```

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.

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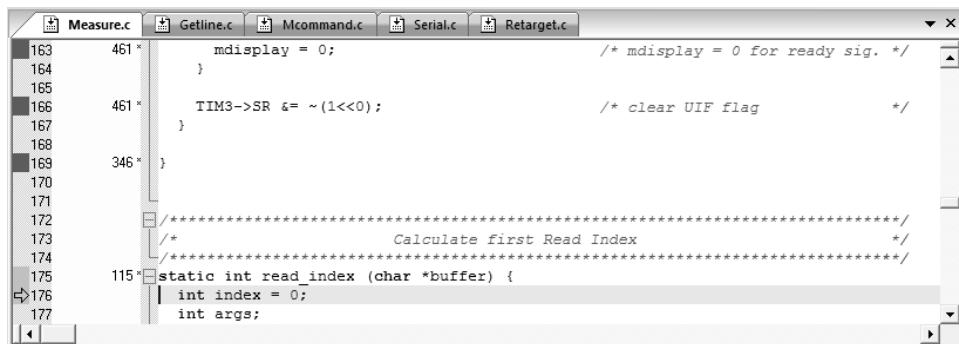
[1] *You can add, remove, and change Toolbox buttons at any time. Use the Command Line in the Command Window for this purpose.*

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.
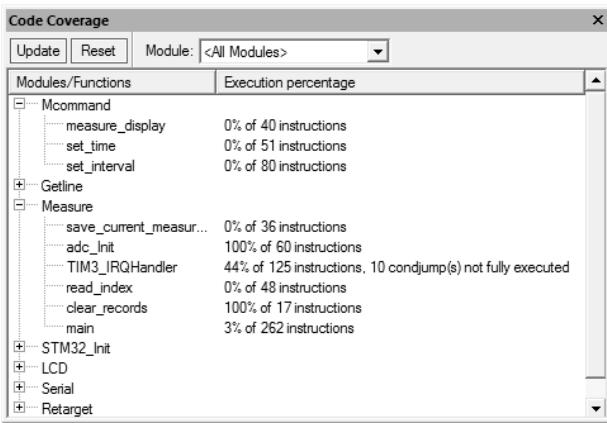
Code Coverage

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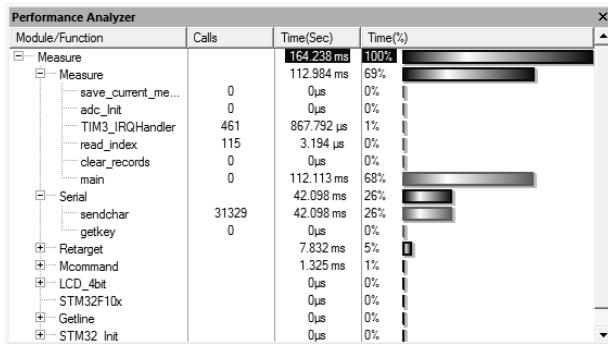
In addition to the **Code Coverage Window**, µVision provides color-coded hints on the side bar of the **Disassembly** and **Editor Window**. The colors have the following meaning:

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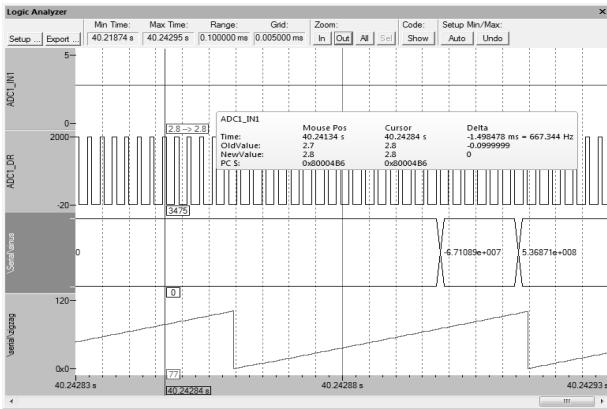
Double-click an object exposed in the Module/Function column to jump to the source code line.

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**.
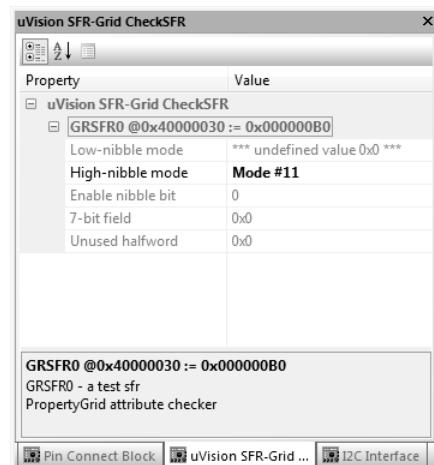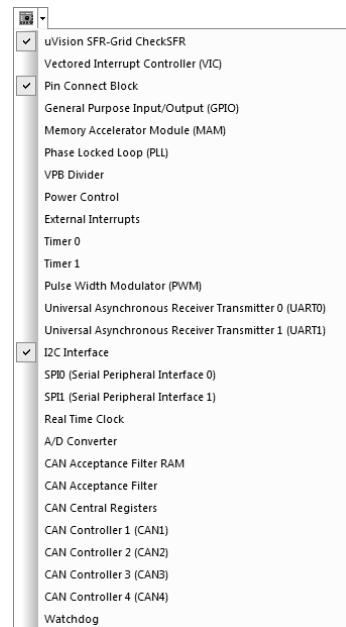
System Viewer

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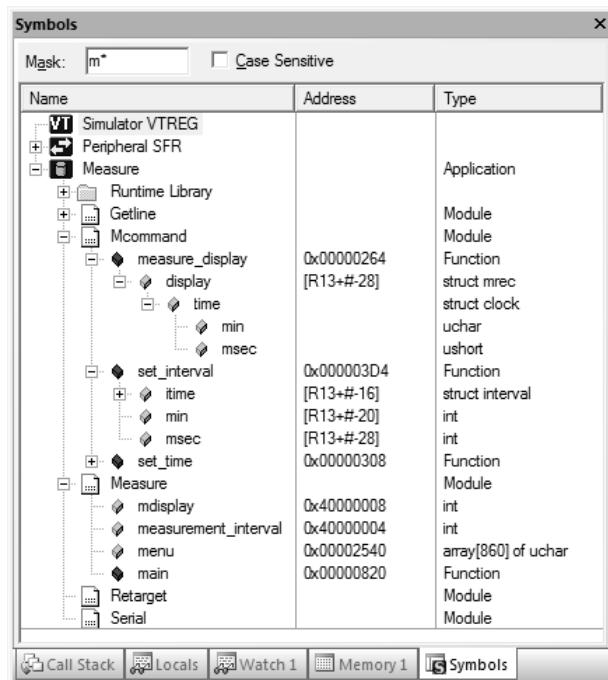
Mask works similar to a find function. Enter your search criteria and browse the results. For nested objects, the entire tree is displayed if a leaf item is found. The following search criteria may be used:

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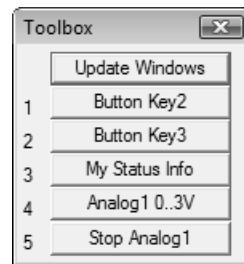
```
KILL BUTTON 5       /* resembles to: "Remove the 'Stop Analog1' button" */
```

---

[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.*

Instruction Trace Window

To follow the instruction sequence history, invoke the I**nstruction Trace Window** from the **Debug Toolbar** or via the **View – Trace** Menu. Use this window in conjunction with the **Disassembly Window**. Trace recording has to be enabled to gather the information needed. To do so, use the **View – Trace – Enable Trace Recording** Menu.

Double-click any line in the **Instruction Trace Window** to jump to or open the **Disassembly Window**. Use the predefined **Filter** options to view the instruction tree in the preferred mode.

This functionality is available for the **Simulator** and while debugging the target hardware. The window's look and feel might vary, since it depends on the driver settings of the debugging environment.
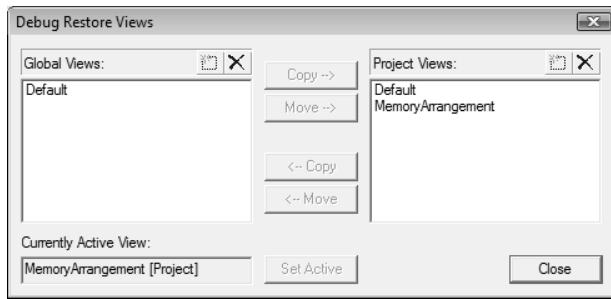
Defining Debug Restore Views

Multiple window layouts are possible in **Debug Mode** to switch quickly between preferred screen settings and window arrangements. Invoke the layouts from the **Window – Debug Restore Views…** dialog, or from the **Debug Toolbar**. Restore defaults through **Window – Reset View to Defaults**.

Define and save your preferred look and feel through the **Window Restore Views…** dialog.

Global Views propagate to all your projects, where **Project Views** are bound to that particular project.

Chapter 8. Using Target Hardware

This section describes the debugging possibilities of µVision in conjunction with your target hardware. The Keil ULINK USB-JTAG Adapter family is discussed in detail, and third-party adapters are mentioned.

The following device families are supported by the Keil **ULINK** adapters:

- 8051   **ULINK** for Infineon XC8xx, ST µPSD3xxx, and NXP LPC95x

- 166   **ULINK** for Infineon C166, XE166, and XC2000

- ARM   **ULINK**, **ULINK***Pro* for ARM7, ARM9, and Cortex-Mx devices

The µVision Debugger interfaces to the target hardware through the following drivers, which are provided by Keil:

- 8051   **Monitor**, **FlashMon**, **MonADI**, **ISD51**, **EPM900**, **Infineon DAS**

- 251   **Monitor**

- 166   **Monitor** for C166

- 166   **Monitor**, **Infineon DAS** for XE166, XC2000

- ARM   SEGGER **J-Link/J-Trace** for ARM7, ARM9, and Cortex-Mx

In addition, many third-party vendors offer µVision drivers for their hardware, for example:
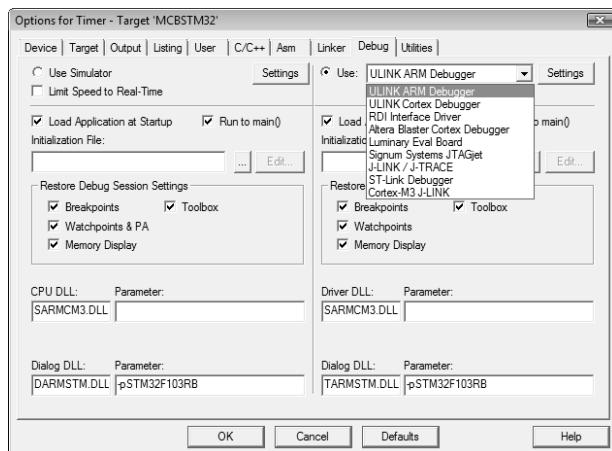
- 8051   Cypress **USB development kits** for EZ-USB devices

- 8051   Quickcore **FPGA based Pro8051** device

- 8051   SST **SoftICE** for FlashFlex51 devices

- 8051   Silabs **Debug Adapter** for C8051Fxxx devices

- ARM   Signum Systems **JTAGjet** for ARM7, ARM9, and Cortex-Mx devices

Configuring the Debugger

Choose **Target Options** – from the **Build Toolbar** and select the **Debug** tab

Alternatively, you can use the **Project – Options for Target** Menu, to open this dialog.
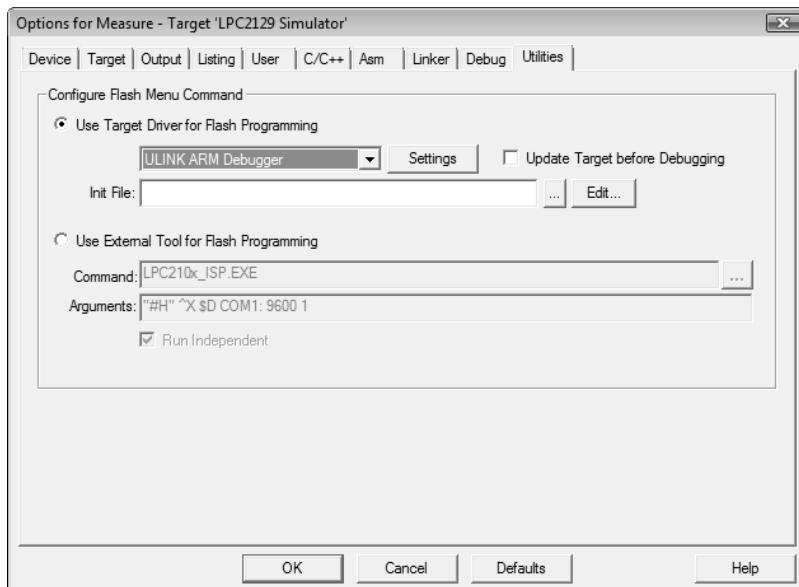
Check the **Use** radio button and select the appropriate debug interface.

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	Limit simulation speed to real-time such that the simulation does not run faster than the target hardware
Run to main()	Program execution halts 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.**

Programming Flash Devices

The µVision IDE can be configured to program the Flash memory of your target system. You can use third-party Flash programming tools that you may attach to and invoke from the development environment. Flash programming is configured from the **Utilities** tab of the **Options for Target** dialog. You have to select the target driver, or a third-party command-line tool, which is usually provided by the chip vendor.

Select **Use Target Driver for Flash Programming** to use a target adapter, like the Keil ULINK USB-JTAG Adapter, SEGGER J-Link, EPM900 Emulator, or Silabs adapter to program your system's Flash memory.

Select **Use External Tool for Flash Programming** to use a third-party command-line utility, like FlashMagic, to program your system's Flash memory.

LOAD   Once configured, the **Download to Flash** button of the **Build Toolbar** or **Flash** Menu downloads the program to your target system's Flash memory
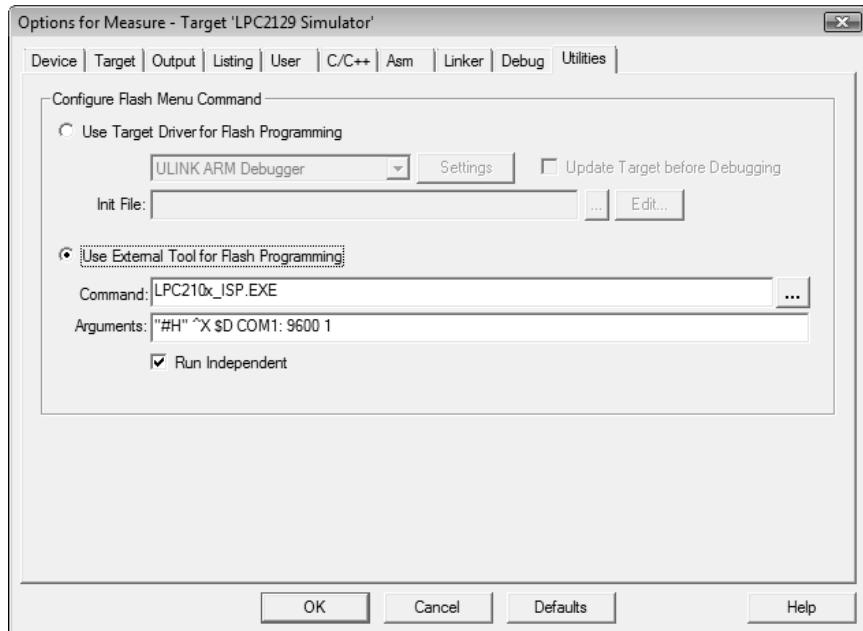
You can configure the µVision Debugger to automatically download to flash memory. To do so, check **Update Target before Debugging**.

Configuring External Tools

To configure µVision for Flash[1] programming with a command-line utility, select
Use External Tool for Flash Programming and specify the **Command** and the
Arguments to be used.

    🛠   Choose **Target Options** – from the **Build Toolbar** and select the
            Utilities tab

Alternatively, you can use the **Project – Options for Target** Menu to open the
Utilities dialog.

Project-specific items, like the path for the generated HEX file, output file name,
device name, or clock frequency can be used in the Arguments field.

Please use the on-line **Help** for additional information.

---
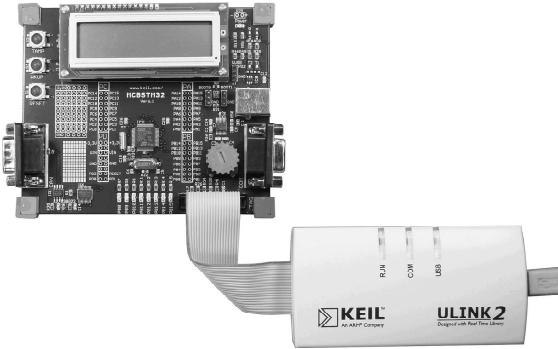
[1] *The µVision Device Database provides the correct configuration for memory Flash of many
microcontroller devices.*

Using ULINK Adapters

The Keil ULINK USB-JTAG family of adapters, further referred to as ULINK, connects your PC's USB port to your target system. The connection between the microcontroller and the ULINK unit can be established via the JTAG[1] port pins of the embedded system. The ULINK adapters enables you to:

- Download target programs

- Examine memory and registers

- Single-step through programs

- Insert multiple breakpoints

- Run programs in real-time

- Program Memory Flash

Before using the Debugger on target hardware, you have to configure the µVision IDE to use the ULINK adapter, or any other external tool suited for Flash programming.

The µVision Debugger can display memory contents and variables in several familiar formats. Memory and variables are updated periodically, providing an instant view of the current program status, even during program execution. It is possible to set breakpoints that trigger on accessing a specific variable.

The Keil ULINK adapter family supports Flash device programming with configurable programming algorithms. You can choose from preconfigured programming algorithms, or customize the algorithms according to your needs. External Flash memory programming is supported for many target systems as well.

---

[1] *The ULINK adapters support a wide variety of devices and protocols, and support your target hardware port pin characteristics.*

ULINK Feature Comparison

Feature	ULINK2	ULINK*Pro*
Run control debug (ARM & Cortex-Mx)	Yes	Yes
Run control debug (8051 & C166)	Yes	-
Data Trace(Coretex-M3)	Yes	Yes
Instruction Trace(Cortex-M3)	-	Yes
JTAG Clock Speed	10MHz	50MHz
Flash Download	28 KBytes/s	600 KBytes/s

Configuring µVision for ULINK Adapters

When using ULINK adapters, you must change a few settings, so that the µVision IDE knows how to use the ULINK adapters for debugging. In detail, you must configure:

- Debug Settings
- Trace Settings (for Cortex-Mx devices only)
- Flash Download

Connect the ULINK adapter to your PC. Only then, the ULINK configuration is possible in µVision.
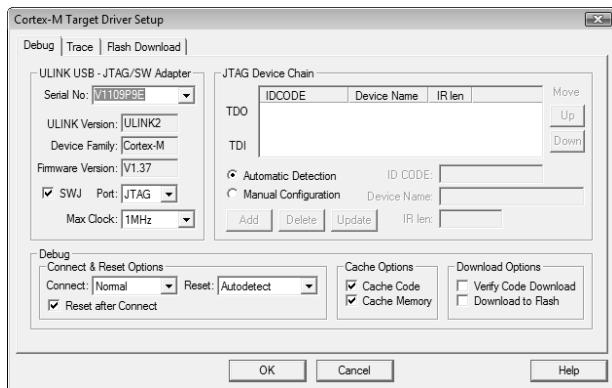
Click **Target Options** from the **Build Toolbar** and select the **Debug** tab, or open the dialog from the **Project – Options for Target – Debug** Menu

Click the **Settings** button to open the **Target Driver Setup** dialog.

Configuring Debug Settings

The **Target Driver Setup**
dialog depends on the
target device selected in
your project.

Please use the on-line **Help**
for additional information.
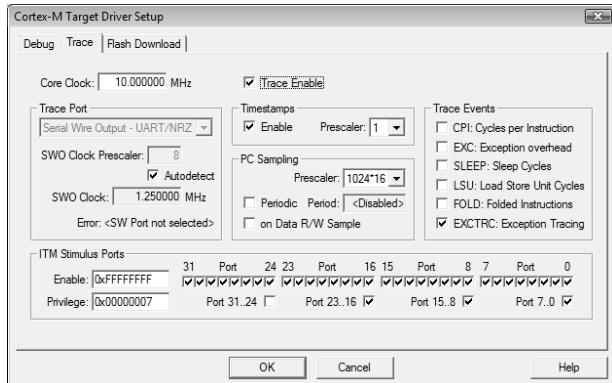
Configuring Trace Settings

The **Trace** dialog tab
controls the real-time trace
operations.

Please use the on-line **Help**
for additional information.

The **Trace** features are
available for Cortex-Mx
devices only.

Configuring Flash Download

The Keil ULINK drivers support a wide variety of Flash-based microcontrollers.
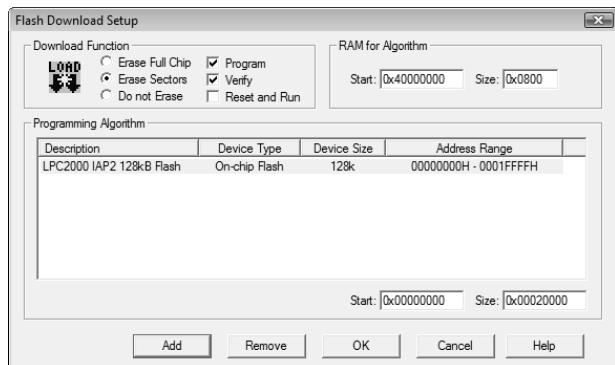
Click **Target Options** from the **Build Toolbar** and select the **Utilities** tab, or open the dialog from the **Project – Options for Target** Menu

To configure µVision for a specific driver, select **Use Target Driver for Flash Programming** and choose the appropriate driver from the drop-down control.

Use the **Settings** button to open the driver-specific **Flash Download Setup** dialog.

Here, you can configure how Flash Download works and specify the programming algorithms that are required by your target system.

Please use the on-line **Help** for additional information.

Programming Algorithms

The ULINK driver allows
you to specify the
programming algorithm
used to program Flash
memory.

Add Flash Programming Algorithm			
Description	Device Type	Device Size	▲
ADuC702X Flash (v1.1)	On-chip Flash	62k	≡
ADuC703X Flash 32KB (v1.4)	On-chip Flash	30k	
ADuC703X Flash 64KB (v1.1)	On-chip Flash	62k	
ADuC703x Flash 96KB (v1.4)	On-chip Flash	94k	
ADuC706X Flash 32KB (v1.0)	On-chip Flash	30k	
ADuC712x Flash (v1.2)	On-chip Flash	126k	
ADuC7229 Flash (v1.2)	On-chip Flash	126k	
AM29F160DB Flash	Ext. Flash 16-bit	2M	
AM29F160DT Flash	Ext. Flash 16-bit	2M	
AM29F320DB Flash	Ext. Flash 16-bit	4M	
AM29F320DB Dual Flash	Ext. Flash 32-bit	8M	
AM29F320DT Flash	Ext. Flash 16-bit	4M	
AM29F320DT Dual Flash	Ext. Flash 32-bit	8M	
AM29x033 Flash	Ext. Flash 8-bit	4M	
AM29x128 Flash	Ext. Flash 16-bit	16M	
AM29x800BB Flash	Ext. Flash 16-bit	1M	▼

Use the **Add** button of the
Flash Download Setup
dialog to open this dialog
**Add Flash Programming
Algorithm**.
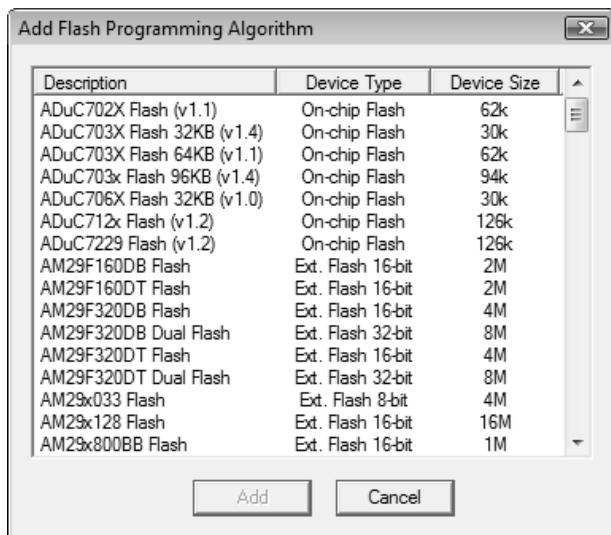
From here you can select
one or more programming
algorithms, one for each
different Flash device.
Highlight your preferred
programming algorithm to **Add** it for your target hardware.

If the Flash device you use is not listed, you may define new algorithms. Do this
for a new Flash device, which is currently not directly supported by Keil. You
may use the algorithms found in any **FLASH** folder as a template for new
algorithms.

The programming algorithms included in your kit are stored in these folders:

- ARM Toolset: \KEIL\ARM\FLASH\
- C16x Toolset: \KEIL\C166\FLASH\

Using an Init File

Some applications or target systems require the execution of specific debug commands or functions ahead of Flash programming. This feature is typically used to define BUS configuration for your device or to program Flash with auxiliary files, containing code or data, which are not included in your target program. The debug commands and functions are stored in an initialization file defined by the **Init File** text box. This file is executed before the Flash download is performed.

BUS Configuration

Typically, the BUS system has to be configured before a device with external Flash memory can be programmed. If you use the ULINK USB-JTAG adapter, you may create an initialization file that uses predefined debug functions, like _**WBYTE** and _**WDWORD**, to write to memory and configure the BUS. For example:

```
_WDWORD(0xFFE00000, 0x20003CE3);   // BCFG0: Flash Bus Configuration
_WDWORD(0xE002C014, 0x0E6001E4);   // PINSEL2: CS0, OE, WE, BLS0..3
```

Auxiliary Memory Content

In addition to BUS configuration, the initialization file may contain instructions to load auxiliary programs or data into memory. For example:

```
LOAD MyFile.HEX
```

By default, the executable specified **Project – Options for Target – Output** is downloaded to Flash.

Chapter 9. Example Programs

Each Keil toolset includes example programs, which are ready to run and which help you to get started. Browse the examples to learn how the development tools work and get familiar with the look and feel, as well as with the behavior of µVision. You may copy the code of the examples for your own purpose.

Example programs[1] are stored in the **\EXAMPLES** folder, where each program resides in a separate subfolder along with its project files. Thus, you can re-build the examples and evaluate the features of µVision quickly.

While there are numerous example programs for you to examine, this manual describes and demonstrates only four:

- Hello: Your First Embedded Program
- Measure: A Remote Measurement System
- Traffic: A Traffic Light and Pedestrian Cross Walk System
- Blinky: An example of how to use target hardware

As described in the previous chapters, many actions or functions of µVision can be called from a toolbar, a menu, or by entering a command in the **Command Window**. Some actions may be triggered through key combinations.

We advise you to try out the various functions of µVision while in **Debug Mode**. Please test the features described in preceding chapters. In particular, get familiar with the navigation, invoke the **Context Menu** of various objects, drag and drop windows to other screen areas or other physical screens, and create and save personalized layouts. Invoke the **Performance Analyzer**, **Logic Analyzer**, **Code Coverage**, **Symbols Window**, and drag and drop items from one window to another window. Single-step through the code, get familiar with the **Disassembly Window**, and inspect how it works in conjunction with the **Register Window**, **Output Window**, and **Serial Window**.

---

[1] *Example programs are license free.*

"Hello" Example Program

The first program in any programming language simply prints "Hello World" to the screen. In an embedded system, there is no screen, so the "Hello" program sends its output to the on-chip serial port. This entire program has one single source file, **HELLO.C**.

This small application helps you to confirm that you can compile, link, and debug an application. You may perform these operations from the command line, using batch files, or from µVision using the provided project file.

The target hardware[1] for the "Hello" project is based on a standard microcontroller. Examples are provided for all supported architectures and are located in the folders as specified in the table below.

Architecture	Example Folder
ARM	**\KEIL\ARM\EXAMPLES\HELLO**
C166/XE166/XC2000	**\KEIL\C166\EXAMPLES\HELLO**
8051	**\KEIL\C51\EXAMPLES\HELLO**

Opening the "Hello" Project

To begin working with the "Hello" project, open the **HELLO.UVPROJ** project file from the appropriate example folder.
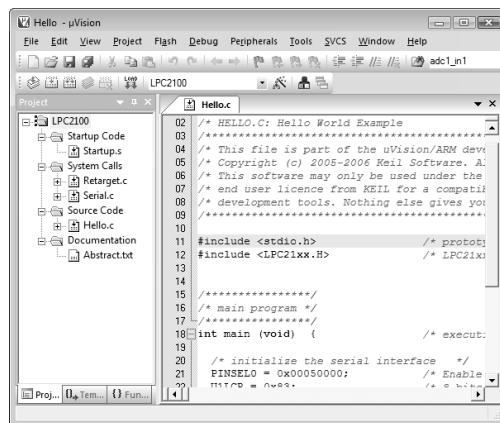
Select the **Project – Open Project** Menu and open **HELLO.UVPROJ** from the ...**\EXAMPLES\HELLO** folder.

Alternatively, you may drag and drop the **HELLO.UVPROJ** file into the µVision application, or simply double-click the file.

---

[1] *Since µVision simulates the target hardware required for this program, you actually do not need target hardware or an evaluation board.*

Once the project has been opened, µVision shows the source files that comprise the project. The files are shown in the **Project Window**. Double-click on **HELLO.C** to view or edit the source file. µVision loads and displays the file contents in the **Editor Window**.
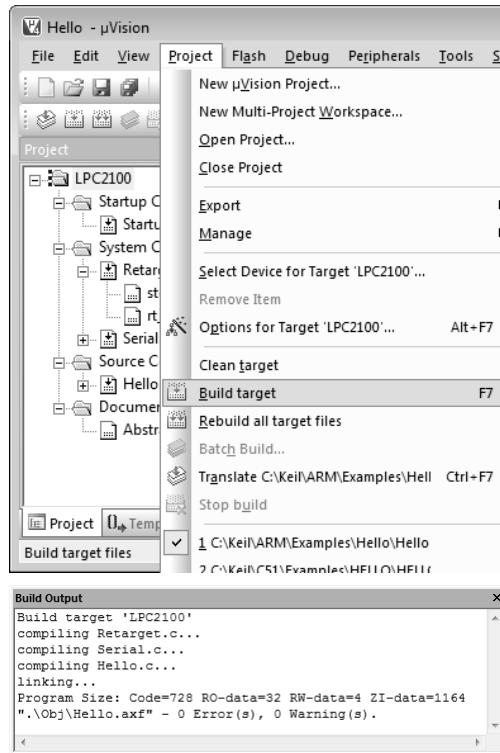
Building the "Hello" Project

Compile and link the project using the **Build** button of the **Build Toolbar**, or select the **Project – Build Target** Menu.

µVision runs the assembler and compiler, to assemble and compile the source files of the project. The linker adds the necessary object modules and combines them into a single executable program, which may be loaded by the µVision Debugger for testing.
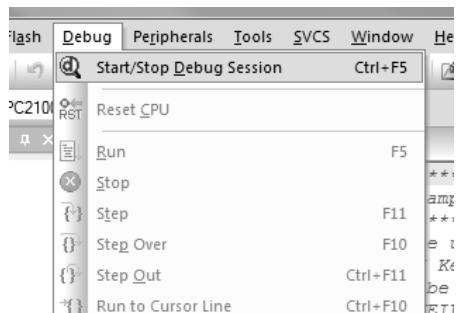
You can follow the build process in the **Build Output Window**. Errors, warnings, and additional trace messages are displayed here. Double-click an error or warning message to jump to the source line that triggered the notification.

Testing the "Hello" Project
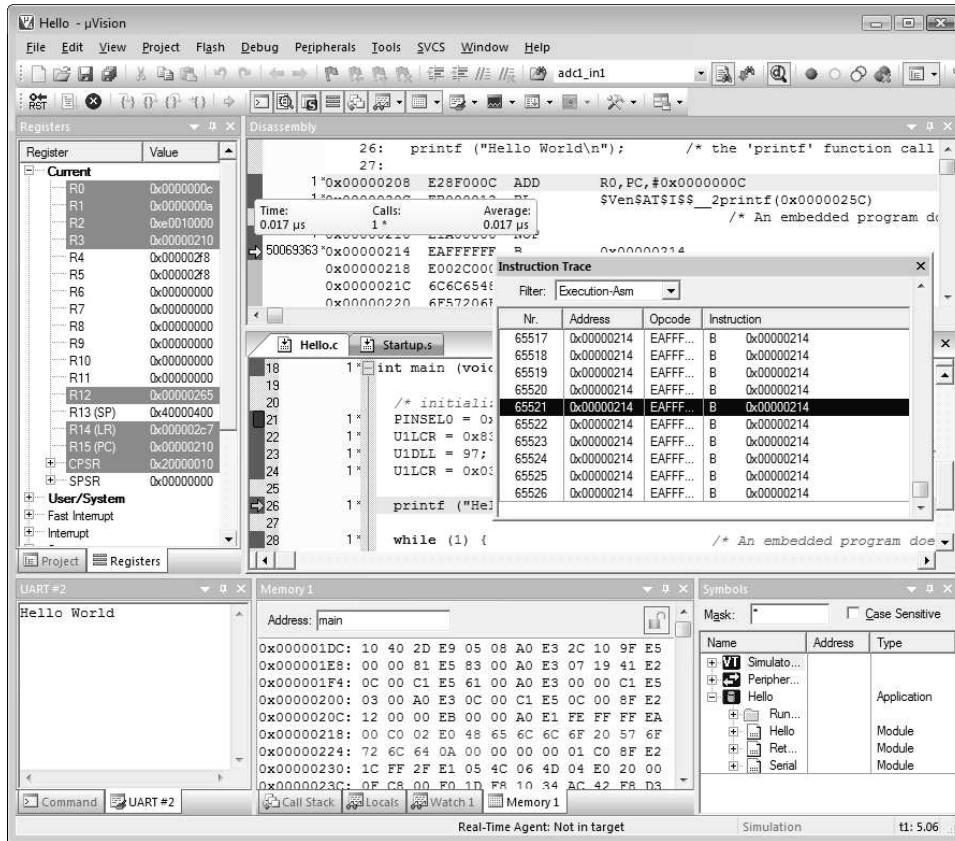
Once the "Hello" program has been compiled and linked successfully, test it with the µVision Debugger. Select **Debug – Start/Stop Debug Session** from the menu or from the **File Toolbar**. µVision initializes the debugger, starts program execution, and halts before entering the main() C function.

Use the following debugger commands to control program execution.

📝 Open the **Serial Window UART #1** to display the application's output

📜 Click the **Run** button of the **Debug Toolbar** or choose **Debug – Run** to start the "Hello" program. "Hello World" is printed to the **Serial Window** and the program enters into an endless loop.

❌ Click the **Stop** button to halt the program. Alternatively, press the **Esc** key while in the **Command Line** of the **Command Window**

⬤ Use the **Insert/Remove Breakpoint** command to set or clear a breakpoint

⚙ Test the **Reset** command to reset the simulated microcontroller. If the program is still running, it halts at the first breakpoint.

↪ Single-step through the program using the **Step** buttons. The current instruction, which will execute next, is marked with a yellow arrow. The yellow arrow moves each time you step.

While debugging, µVision displays the following default screen layout. If you
re-arrange the layout, µVision saves the layout automatically and provides this
layout next time you invoke the debugger. However, you cannot explicitly recall
the changed layout, unless you saved it through the **Window – Debug Restore
Views…** Menu.

Take the opportunity to get familiar with the new look and feel, and the
navigation features. See how the content of registers and memory areas can be
changed. Display the values in the different representations. We recommend
taking some time and using this simple example to explore the µVision
capabilities.

"Measure" Example Program

The "Measure" program is a simple example that collects analog and digital data using methods similar to those that can be found in weather stations and process control applications. Three source files: **GETLINE.C**, **MCOMMAND.C**, and **MEASURE.C** are used.

The "Measure" program[1] records data received from digital ports and A/D inputs. A timer, which can be configured between 1 millisecond and 60 minutes, controls the sample rate and interval. The current time and all data from the input channels are measured and saved to a RAM buffer.

Please find your preferred "Measure" program in one of the following locations:
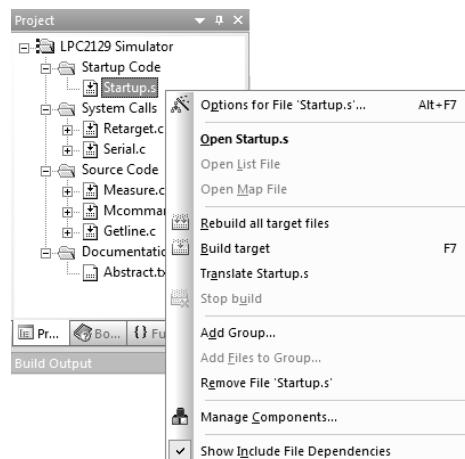
Architecture	Example Folder
ARM	**\KEIL\ARM\EXAMPLES\MEASURE**
C166/XE166/XC2000	**\KEIL\C166\EXAMPLES\MEASURE**
8051	**\KEIL\C51\EXAMPLES\MEASURE**

Opening the "Measure" Project

To start the "Measure" project, open the project file **MEASURE.UVPROJ** from the example folder of your choice.

Compile the program; enable or disable the include files in the project structure, invoke the **Context Menu** of the **Project Window**, and toggle **Show Include File Dependencies**.

Three application-related source code files are located in the **Source Code** group. The serial I/O and system modules are placed in the **System Calls**, whereas the startup file resides under the **Startup Code** group.

---

[1] *Since μVision simulates the hardware required for this program, you do not actually need target hardware or an evaluation board.*

A project may contain one or more targets, a feature that allows you to build different versions of your program. The "Measure" project contains several targets for different test environments including the simulator and evaluation boards. Select the model with the Simulator target. The following files comprise the source code:

MEASURE.C   This file contains the main C function and the interrupt service routine for the timer. The main function initializes all peripherals and performs command processing. The interrupt routine manages the real-time clock and sampling.

MCOMMAND.C   This file processes the display, time, and interval commands. These functions are called from the main C function. The display command lists the analog values in floating-point format to give a voltage between 0.00V and 3.00V.

GETLINE.C   This file contains the command-line editor for characters received from the serial port.

Building the "Measure" Project

There are several commands you can access from the **Project** Menu and the **Build Toolbar** to compile and link the files in a project.

- Use the **Translate File** command to compile the selected file in the Project Workspace

- Use the **Build Target** command to compile files that have changed since the last build and link them

- Use **Rebuild All Target Files** command to compile and link all files in the project

- Use the **Stop Build** command to halt a build that is in progress

Select the **Build Target** command to compile and link the source files of the "Measure" project. µVision displays a message in the **Command Window** when the build process has finished.

Source Browser

The "Measure" project is configured to generate complete browser and debug information.

📝   Use the **Source Browse** command from the **File Toolbar** or **View** Menu to view information about program variables and other objects

Testing the "Measure" Project

The "Measure" program is designed to accept commands from the on-chip serial port. If you have actual target hardware, you may use Hyperterm or another terminal program to communicate with the board. If you do not have target hardware, you can use µVision to simulate all aspects of the hardware, for example, the **Serial Window** in µVision simulates serial input.

🔍   Use the **Start/Stop Debug Session** command from the **Debug Toolbar** or **Debug** Menu to start the µVision debugger

Using the Serial Commands
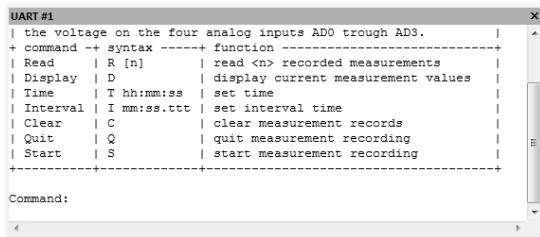
Test the following commands in the **Serial Window**.

Action	Command	Description
Clear	C	Clears the measurement record buffer
Display	D	Displays the current time and input values continuously
Interval	I *mm:ss.ttt*	Sets the time interval for measurement recording. The interval time must be between 0:00.001 (for 1ms) and 60:00.000 (for 60 minutes).
Quit	Q	Quits the measurement recording
Read	R [*count*]	Displays the saved records. Specify the number of records to be shown. All records are transmitted if count is not specified. You can read records on the fly if the interval is greater than one second, otherwise recording must be stopped.
Start	S	Start recording. Data inputs are stored at the specified time interval.
Time	T *hh:mm:ss*	Sets the current time in 24-hour format

Using the Serial Interface

📝   Open the serial **UART Window** from the **View** Menu or the **Debug Toolbar** to view the output

Before you start running the "Measure" program, open the Serial Window so that you can enter commands and view the program output.

```
UART #1                                                                  ×
| the voltage on the four analog inputs AD0 trough AD3.           |  ▲
+ command -+ syntax -----+ function --------------------------+
| Read     | R [n]       | read <n> recorded measurements      |
| Display  | D           | display current measurement values  |
| Time     | T hh:mm:ss  | set time                            |
| Interval | I mm:ss.ttt | set interval time                   |
| Clear    | C           | clear measurement records           |
| Quit     | Q           | quit measurement recording          |
| Start    | S           | start measurement recording         |
+----------+-------------+-------------------------------------+

Command:                                                                 ▼
◄                                                                        ►
```

Running the Program

Use the step-buttons to execute code commands individually. If the **Disassembly Window** is the active window, the debugger steps through assembler instructions rather than through the source code.
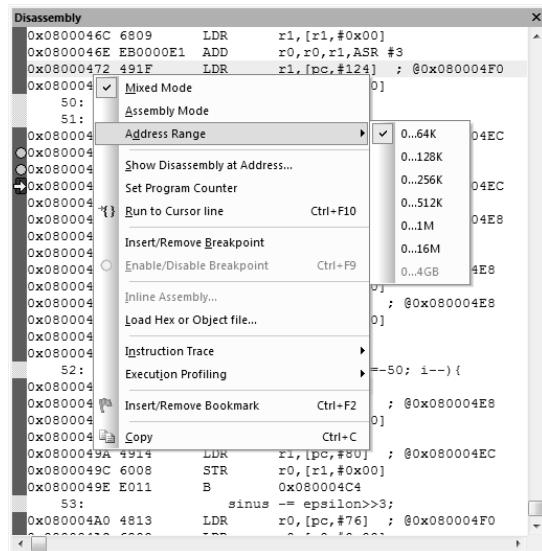
⇨   The current instruction or high-level statement (the one about to execute) is marked with a yellow arrow. Each time you step, the arrow moves to reflect the new current instruction line.

📋   Use the **Run** command from the **Debug Toolbar** or **Debug** Menu to start debugging the program

❌   Use the **Stop** command to halt program execution or press the **Esc** key while in the **Command Window**

�{ }   Use the **Step Into** command from the **Debug Toolbar** or **Debug** Menu to step through the program and into function calls

〔〕   Use the **Step Over** command from the **Debug Toolbar** or **Debug** Menu to step through the program and over a function call

〔〕   Use the **Step Out** command from the **Debug Toolbar** or **Debug** Menu to step out of the current function

*{ }   Use the **Run To Cursor Line** command from the **Debug Toolbar** or **Debug** Menu to run the program to the line you just highlighted

Viewing Program Code

Use the **Disassembly Window** command from the **Debug Toolbar** or **View** Menu to view mixed source and assembly code

Test the various stepping commands, first while in the **Disassembly Window**, and then while in the **Editor Window**. Notice the different behavior of the Debugger.
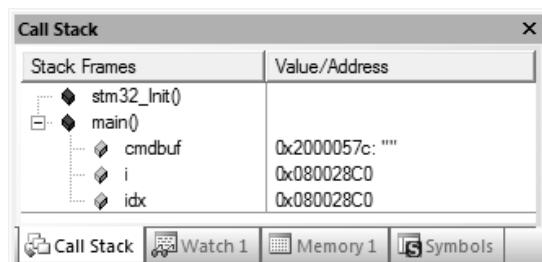
Call the **Context Menu** of the windows while moving the mouse over various code lines. Notice the different options. They depend on whether the statement can be executed or not. Notice the lines marked green, gray, or without any color. Invoke the **Context Menu** while pointing on the memory address.

Using the Call Stack

Use the **Call Stack Window** command of the **Debug Toolbar** or **View** Menu
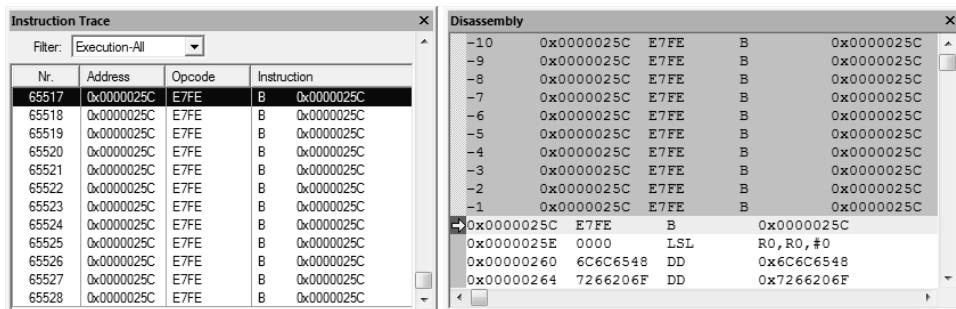
µVision tracks function nesting and displays data in the **Call Stack Window**. Double-click on the line of a function to jump to the source code.

Using the Trace Buffer

In any programming process, it is often required to investigate circumstances that led to a certain state. You can guide the µVision Debugger to record instructions into a trace memory buffer. In **Debug Mode**, you can review the trace buffer using the **View – Trace – Show Records in Disassembly** command.

    Use the **Trace Windows** command from the **Debug Toolbar** or from the **View – Trace – Instruction Trace Window** Menu to view executed instructions stored in the trace buffer

Whereas the trace information is always available in the **Disassembly Window**, the **Instruction Trace Window** is enabled for ARM devices only.

In addition, inspect the **Registers Windows** showing register contents of the selected instruction.

Invoke the **Context Menu** of the **Disassembly Window** to review the options offered.

When you double-click in the **Instruction Trace Window**, the **Disassembly Window** shows the corresponding instruction.

Using Breakpoints
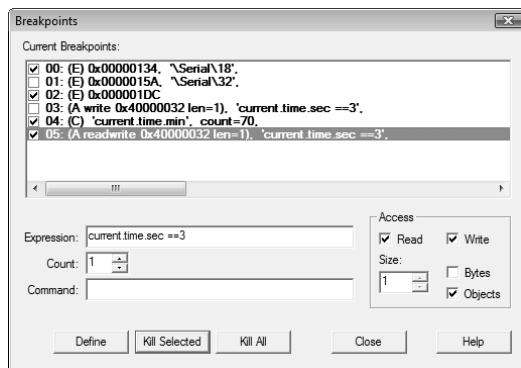
µVision supports execution, access, and complex breakpoints. The following example shows how to create a breakpoint that is triggered when the value 3 is written to **current.time.sec**.

Open the **Breakpoints** dialog from the **Debug – Breakpoints** Menu. Enter the **Expression** **current.time.sec==3** and select the **Write** check box. This specifies the breakpoint to trigger when the program writes the value 3 to **current.time.sec**. Click the **Define Button** to set the breakpoint. Double-click any breakpoint definition to redefine it.

Reset the CPU to test the breakpoint, which will trigger and halt program execution when the number 3 is written to **current.time.sec**. The program counter line of the **Debug Window** marks the position where the breakpoint triggered.
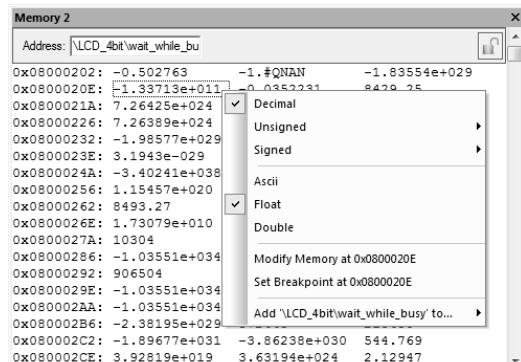
Viewing Memory Contents

Use the **Memory Window** command from the **Debug Toolbar** or **View** Menu to display the memory content

Use the **Lock/Freeze** icon to prevent values from refreshing

µVision displays memory in various formats and reserves four distinct **Memory Windows**.

Define the starting **Address** to view the content, or drag and drop objects from the **Symbols Window** into the **Memory Window**.

Open the **Context Menu** to change formats, modify memory, or set breakpoints.

Watching Variables

The µVision Debugger provides two **Watch Windows** to which you can add variables, structures, and arrays for easy reference. The **Watch Window** updates at the end of each execution command. Enable the **View – Periodic Window Update** Menu to refresh the content of this window during program execution.

> Use the **Watch Window** command on the **Debug Toolbar** or **View** Menu to launch the functionality or invoke the **Locals**

The **Locals** page shows local symbols of the currently executed function.

The **Watch** pages display program objects, which you desire to monitor. Structures and arrays open on demand when you click on the [**+**] symbol. Indented lines reflect nesting.

Name	Value
\Measure\mdisplay	0x00000000
(PORT0 & 0x8000)>>15	0x00000001
`Port0	0x7BFFFFFF
\Measure\setinterval	struct interval { ... }
min	0x00
sec	0x00
msec	0x0000
`CLOCK	0x10642AC0
<double-click or F2 to add>	

Watch 1 — Locals / Watch 1

There are several ways to add variables to **Watch** pages.

▪ In the **Watch Window**, select the last line (**<double-click or F2 to add>**) on the Watch page. Press **F2** or click with the mouse on this line. Enter the name of the variable you wish to watch.

▪ Select a variable in the **Editor Window**, open the **Context Menu** (right-click), and select **Add to Watch Window**

▪ In the **Command** page of the **Output Window** enter **WS** (for WatchSet) followed by the **Watch Window** number (1 or 2) followed by the variable name

▪ Simply drag and drop an object into this window

Remove a variable from the **Watch Window** by selecting the line and press the **Del** key or use the **Context Menu**. Individual elements of structures and arrays cannot be removed singly.

Viewing and Changing On-Chip Peripherals
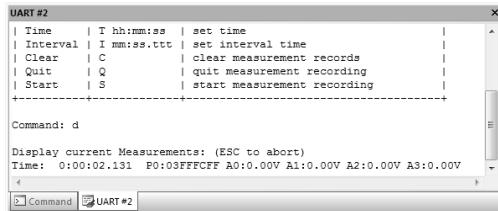
The "Measure" program accepts input from several I/O and A/D ports. Use the µVision Debugger to view data and interact with peripherals. Changes made to the inputs are reflected in the dialog window of each peripheral. Enter **D** in the **Serial Window** to monitor the output and the changes applied to input values.

🔃 **Reset** the simulated CPU

📥 **Start** the program if it is not running already

📝 **Open** the **Serial Window** if it is closed

The **D** command causes the "Measure" program to refresh the time, I/O Ports, and the A/D Inputs continuously. The input from the I/O Port and the A/D converter channels can be controlled from peripheral dialogs, which are available from the **Peripherals** Menu.
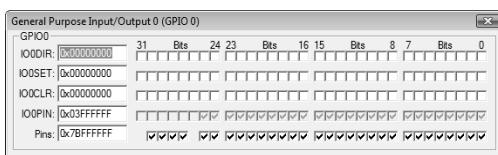
Using Peripheral Windows
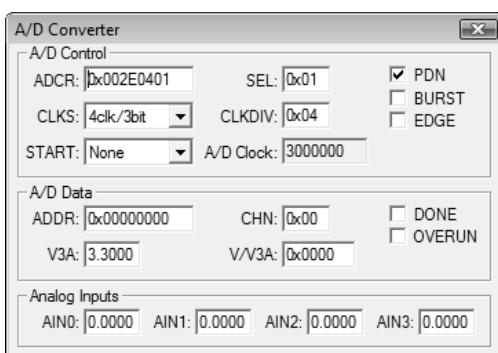
The µVision Debugger provides windows for I/O and serial ports, A/D converters, interrupts, timers, and for most other chip-specific peripherals.
Open the windows from the **Peripherals** Menu.

The windows display the status of the registers as well as the pins of the simulated device.

Open the A/D Converter dialog to view the status of the A/D controls and A/D data. You can enter input voltages for the Analog Input, which are reflected in the Serial Window.

Using VTREG Symbols

In addition to the peripheral dialogs, you may use Virtual Target Registers
(VTREG) to change input signals. On the **Command Window,** you can assign
values to VTREG symbols. For example:

```
PORT0=0xAA55        /* Set digital input PORT to 0xAA55   */
AIN1=3.3            /* Set analog input AIN1 to 3.3 volts */
```

Using User and Signal Functions

The µVision Debugger supports a C-like script language that enables you to use
VTREG symbols in a more programmatic way. A debug signal function is
included in the "Measure" program. It can be invoked using the buttons in the
Toolbox. Inspect the **Command Window** and press **F1** to invoke the on-line
help for further information.
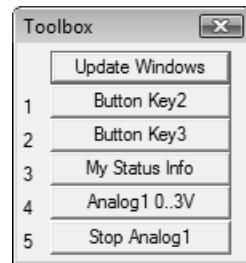
Using the Toolbox

Use the **Toolbox** command from the **Debug Toolbar** or **View** Menu to
display the **Toolbox** dialog

The **Toolbox** contains user-defined buttons that are linked
to debugger commands or to user-defined functions.
Several buttons are predefined for the "Measure" program.

The **Analog0..3V** button starts a user-defined signal
function that provides input to Analog Input 1 on the
simulated microcontroller.

Using the Logic Analyzer

The µVision Debugger includes a configurable **Logic Analyzer** you can use to trace simulated signals and variables during program execution.
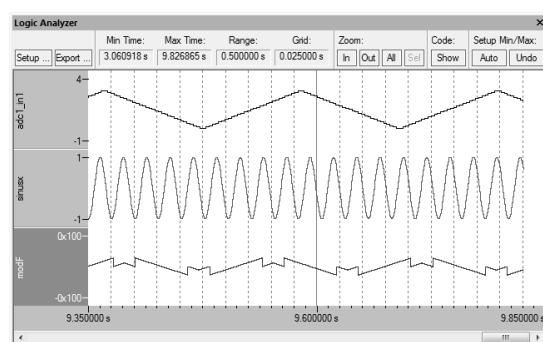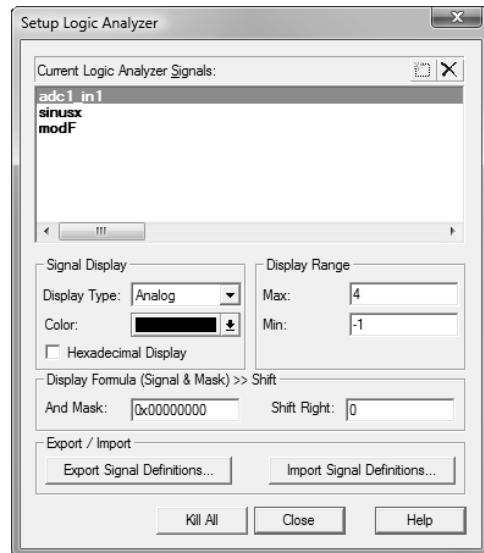
Open the **Logic Analyzer Window**
from the **Debug Toolbar** or **View** Menu

Add a number of signals to the **Logic Analyzer**, including the simulated A/D input signal.

Click the **Setup** button in the **Logic Analyzer Window** to open the **Setup** dialog. Press the **Ins** key, enter **ADC1_IN1**[1], which is the name of the input signal for A/D Channel 1, and close the **Setup** dialog. You may prefer to just drag and drop the object from the **Symbols** Window into the **Logic Analyzer** Window.

For complex analysis, multiple signals can be selected and recorded. Save and load the signal definitions using the **Export Signal Definitions…** and **Import Signal Definitions…** button.

Run the program, and use the **Analog1 0..3** button of the **Toolbox** to start changing the signal on Analog Input 1. Changes applied to the analog inputs are reflected in the **Logic Analyzer**.

---

[1] *The additional signals used in this screenshot are not integrated into the Measure example.*

"Traffic" Example Program

The "Traffic" program[1] is an example that shows how a real-time operating system can be used in an embedded application. This example simulates the control of a traffic light and walk signal. During rush hours, the stop signal controls the traffic flow at an intersection and allows pedestrians to cross the street periodically or by pressing the "**Push for Walk**" button. After rush hours, the traffic light flashes yellow.

You interface the "Traffic" program via the:

- Serial **UART Window**, where you can change the current time and the hours of operation

- **Toolbox**, where you can click the "**Push for Walk**" button to cross the street

- **Watch Window** and I/O Port dialog, where you can watch the state of the traffic light and the start/stop pedestrian lights
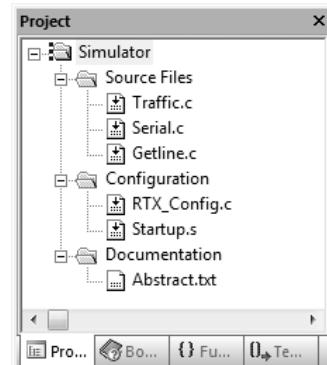
The following table specifies the location of the "Traffic" project files for the various architectures.

Architecture	Example Folder
ARM	\KEIL\ARM\RL\RTX\EXAMPLES\TRAFFIC\
C16x/XC16x	\KEIL\C166\EXAMPLES\TRAFFIC\
8051	\KEIL\C51\RTXTiny2\EXAMPLES\TRAFFIC\

Opening the "Traffic" Project

To start working with the "Traffic" project, open the **TRAFFIC.UVPROJ** project file from the appropriate example folder.

Most Keil example projects include a text file named **ABSTRACT.TXT** that explains the aspects and the intention of the program and is included in the **Project Window**.

---
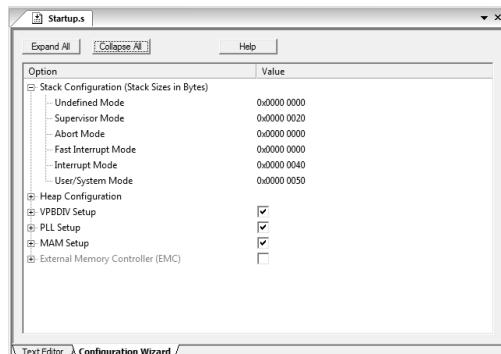
[1] *Since µVision simulates the hardware required for this program, you do not need any target hardware, an evaluation board, or a traffic light.*

Using the Configuration Wizard

µVision incorporates a **Configuration Wizard** that assists you in choosing the settings for the startup file and other configuration files.

Traditionally, these files are assembler or other source files, which include macros or definitions you may change depending on your hardware configuration or preferences.

The **Configuration Wizard** simplifies the process of making these selections.

Of course, you may always edit these files in their original source form by clicking on the **Text Editor** tab.

Building and Testing the "Traffic" Project

Use the **Rebuild** command to compile and link all files of the project
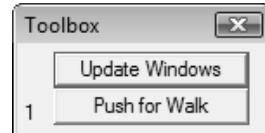
The "Traffic" program is designed to accept commands from the on-chip serial port, which is completely simulated within µVision, and to display output on a traffic light, which is connected to I/O port pins.

Using the Toolbox

Use the **Toolbox** command from the **Debug Toolbar** or **View** Menu to display the toolbox dialog

The **Push for Walk** button is available on the **Toolbox**. Click this button to simulate a pedestrian who wants to cross the road and watch as the "stop" and "walk" lights change in the **Watch Window**.

Using the Watch Window
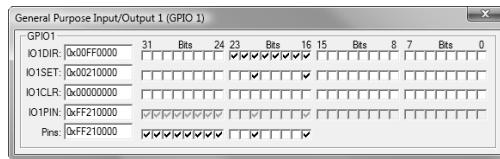
🖼   Use the **Watch Window** command on the **Debug Toolbar** or **View**
     Menu. Open the **Call Stack Window** as well.

The status of the traffic pedestrian
lights displays in the **Watch 1** page
of the **Output Windows** using
predefined watch expressions.

Using I/O Ports

You may also view the traffic light
signal lines on the I/O Port dialog
available from the **Peripherals** Menu.

Using the Serial Window

The **Serial Window** displays
information and allows you to
change the time and operating hours
of the traffic light. Set the current
time outside of the rush hours to
view the flashing yellow light.
Check the **Watch Window** to
monitor the changed behavior.
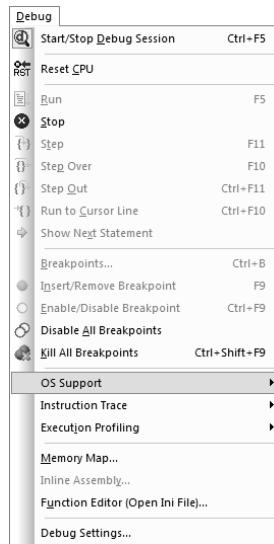
Use the serial commands listed in the following table. These commands are plain
ASCII text. Each command must be terminated with a carriage return.

Action	Command	Description
Display	D	Displays the current time and the Start Time and End Time of operating hours. Press **Esc** to exit display mode.
Set Current Time	T *hh:mm:ss*	Sets the current time for the traffic light. If the current time is within the operating hours specified by the Start and End times, the traffic light operates as normal. If the current time is outside the operating hours, the traffic light flashes the yellow light.
Set Start Time	S *hh:mm:ss*	Sets the Start Time for normal operation
Set End Time	E *hh:mm:ss*	Sets the End Time for normal operation
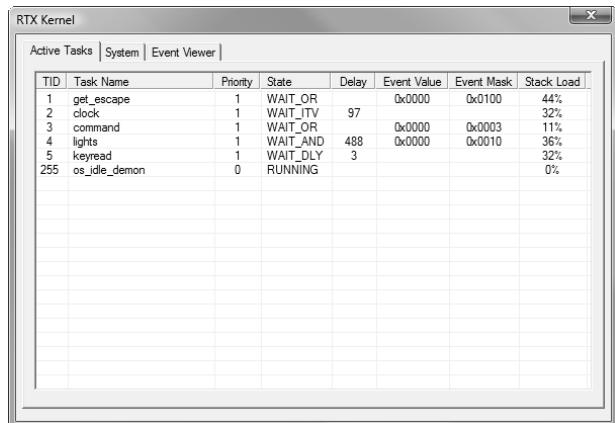
Displaying Kernel-Aware Debug Information

The μVision Simulator allows you to run and test applications created with a real-time operating system. Real-time applications load exactly like other programs. No special commands or options are required for debugging.

Kernel-aware debugging is available in the form of a dialog that displays the aspects of the real-time kernel and the tasks in your program. This dialog can be used with target hardware.

Debug		
Start/Stop Debug Session	Ctrl+F5
Reset CPU	
Run	F5
Stop	
Step	F11
Step Over	F10
Step Out	Ctrl+F11
Run to Cursor Line	Ctrl+F10
Show Next Statement	
Breakpoints...	Ctrl+B
Insert/Remove Breakpoint	F9
Enable/Disable Breakpoint	Ctrl+F9
Disable All Breakpoints	
Kill All Breakpoints	Ctrl+Shift+F9
OS Support	▶
Instruction Trace	▶
Execution Profiling	▶
Memory Map...	
Inline Assembly...	
Function Editor (Open Ini File)...	
Debug Settings...	

To open the kernel-aware debug window, use the **Debug – OS Support** Menu.

RTX Kernel

Active Tasks | System | Event Viewer

TID	Task Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
1	get_escape	1	WAIT_OR		0x0000	0x0100	44%
2	clock	1	WAIT_ITV	97			32%
3	command	1	WAIT_OR		0x0000	0x0003	11%
4	lights	1	WAIT_AND	488	0x0000	0x0010	36%
5	keyread	1	WAIT_DLY	3			32%
255	os_idle_demon	0	RUNNING				0%

"Blinky" Example Program

The "Blinky" program is an example application that blinks LEDs on an evaluation board. The blinking LEDs make it easy to verify that the program loads and executes properly on target hardware.

The "Blinky" program is a board-specific application, and thus, since the boards are different, the program may show other board-specific features. Refer to the board manual for detailed information.
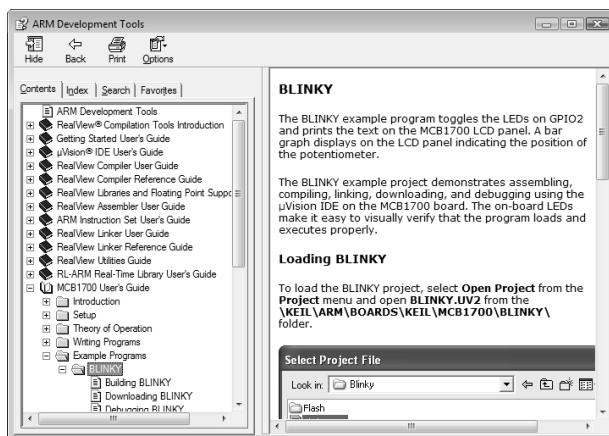
Opening the "Blinky" Project

Select the **Project – Open Project** Menu and choose the respective **BLINKY.UVPROJ** project from the following subfolders:

Architecture	Example Folder
ARM	**\KEIL\ARM\BOARDS*vendor**board name*\BLINKY**
C166/XE166/XC2000	**\KEIL\C166\BOARDS*board name*\BLINKY**
8051	**\KEIL\C51\EXAMPLES\BLINKY**

Each project contains an **ABSTRACT.TXT** file that explains how to use the "Blinky" program for that specific board.

You will also find a detailed description of the "Blinky" program in the **User's Guide** manual of the board.
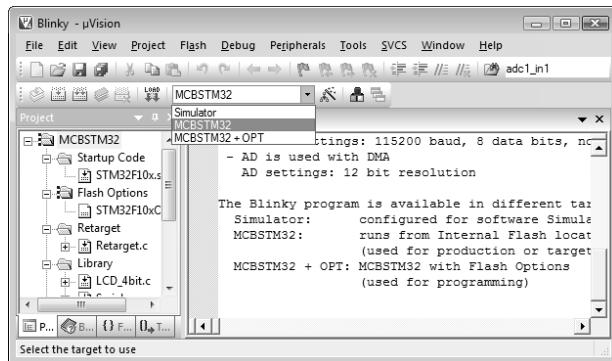
Building the "Blinky" Project

The project may contain several targets, for example:
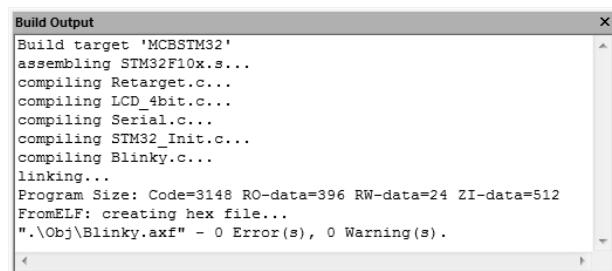
- **Simulator**: is a configuration to debug code without real target hardware

- *Board specific target*: is a configuration to download and test the program on real target hardware

For testing on hardware ensure that *Board specific target* is selected.

🔲 Use the **Rebuild** command of the **Build Toolbar** to compile and link all project files, or use the **Project – Rebuild all target files** Menu

The executable files are placed in an output folder and are ready for downloading.

Downloading the "Blinky" Program

Before using the Download command for the first time, verify the Flash options in the **Project – Options for Target – Utilities** dialog. Alternatively, you can use the **Flash – Configure Flash Tools…** Menu to reach the identical dialog. Connect the board to your PC.

After you have configured the µVision IDE, the **Flash – Download** Menu uses the specified adapter for Flash programming.

LOAD   Click the **Download to Flash** Toolbar button, or use the **Flash –**
         Download Menu to flash the application program to the target hardware
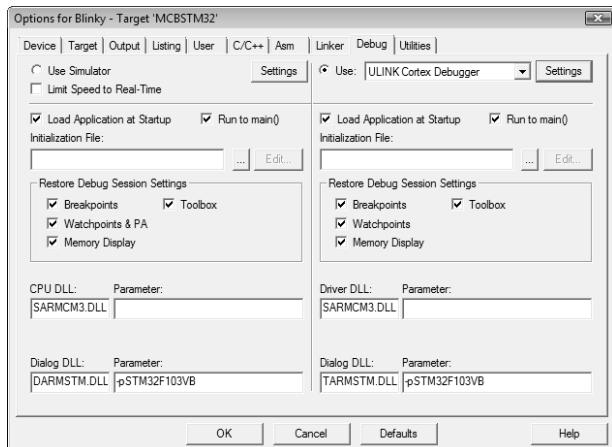
The program has been downloaded, and it runs successfully on your target hardware as soon as the LEDs are blinking.

```
Build Output                                                                  ×
Load "C:\\Keil\\ARM\\Boards\\Keil\\MCBSTM32\\Blinky\\Obj\\Blinky.AXF"
Erase Done.
Programming Done.
Verify OK.
```

Debugging the "Blinky" Program

To verify the configuration settings for the Debugger, open the **Options for Target – Debug** dialog.

- Click the **Use** radio button and select the appropriate debug driver from the drop down list

- Check Load Application at Startup and Run to main()

- Click the **Settings** button and verify the driver configuration

Click **Start/Stop Debug Session** from the **Debug Toolbar**, or open the **Debug – Start/Stop Debug Session** Menu, to start debugging your application

Step One Line – use the step commands to debug the application on target hardware

Reset – Reset the microcontroller while debugging

Run – the program to flash the LEDs on your evaluation board

Stop – program execution

Show Current Statement – Show next statement to be executed in the code

Glossary

ASCII

American Standard Code for Information Interchange
This is a set of codes used by computers to represent digits, characters, punctuation, and other special symbols. The first 128 characters are standardized. The remaining 128 are defined by the implementation.

Assembler

A computer program to create object code by translating assembly instruction – mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities. Programs written in assembly language and translated by an assembler can be loaded into memory and executed.

CAN

Controller Area Network
Is a bus standard, designed specifically for automotive applications, meanwhile also used in other industries. It allows microcontrollers and devices to communicate with each other without a host computer.

CMSIS

Cortex Microcontroller Software Interface Standard
A vendor-independent hardware abstraction layer for the Cortex-Mx processors. It enables consistent, scalable, and simple software interfaces to the processor for interfacing peripherals, real-time operating systems, and middleware, simplifying software re-use, and reducing the time to market for new devices.

Compiler

A program that translates source code from a high-level programming language, such as C/C++, to a lower level language, for example, assembly language or machine code. A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis, code generation, and code optimization. µVision implements C/C++ compilers.

CRC

Cyclic Redundancy Check
Is a type of function to detect accidental alteration of data during transmission or storage.

Debugger

A computer program to test software. Debuggers offer sophisticated functions such as running a program step-by-step (single-stepping), stopping, pausing the program to examine the current state at some kind of event through breakpoints, and tracking the values of variables.

FPGA

Field-Programmable Gate Array
A semiconductor device that can be configured by the customer after manufacturing.

GPIO

General Purpose Input/Output
An interface available on microcontroller devices to interact digitally with the outside world. GPIOs are often arranged into groups, typically of 8, 16, or 32 pins. The GPIO port pins can be configured individually as input or output.

ICE

In-Circuit-Emulator
A hardware device used to debug software of an embedded system. It provides hardware-level run-control and breakpoint features. Some ICEs offer a trace buffer that stores the most recent microcontroller events.

Include file

A text file that is incorporated into a source file using the **#include** preprocessor directive.

Instruction set

An instruction set, or instruction set architecture (ISA), is the part of the microcontroller architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes – the native commands implemented by a particular microcontroller.

JTAG

Joint Test Action Group
The common name used for the IEEE 1149.1 standard called Standard Test Access Port and Boundary-Scan Architecture. JTAG is often used as a microcontroller debug or probing port and allows data transfer out of and into device memory.

Library

Is a file, which stores a number of possibly related object modules. The linker can extract modules from libraries to use them in building an object file.

LIN

Local Interconnect Network
Is a vehicle bus standard or computer networking bus-system used within current automotive network architectures. The LIN bus is a small and slow network system that is used as a cheap sub-network of a CAN bus.

Linker

Is a program that combines libraries and objects, generated by a compiler, into a single executable program.

Lint

A tool to check C/C++ code for bugs, glitches, inconsistency, portability, and whether the code is MISRA compliant.

Macro

Defines a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence.

MDI

An application that allows the user to open more than one document from the same application without having to purposely launch another instance of the application.

Memory model

Is a definition that specifies which memory areas are used for function arguments and local variables.

MISRA

Motor Industry Software Reliability Association
A forum that provides software development standards for the C/C++
programming language, focusing on code safety, portability , and reliability
in the context of embedded systems.

Monitor

Is a program for 8051 and C166 devices. It can be loaded into your target
microcontroller to aid in debugging and rapid product development through
rapid software downloading.

Object

A memory area that can be examined. Usually used when referring to the
memory area associated with a variable or function.

Object file

Created by the compiler, this file contains an organized collection of
objects, which are sequences of instructions in a machine code format, but
might also contain data for use at runtime: relocation information, stack
unwinding information, comments, names of variables and functions for
linking, and debugging information.

OCDS

On Chip Debug Support
A debug port that provides hardware emulation features for the Infineon
166 devices.

Opcode

An operation code is that portion of a machine language instruction that
specifies the operation to be performed. Their specification and format are
laid out in the instruction set architecture of the processor.

Simulation

Is the imitation of a real thing or process. In µVision, simulation is used to
create embedded applications without using 'real' hardware. You can
represent key characteristics and behaviors of the selected system, perform
optimization, safety engineering, testing, and debugging.

Stack

An area of memory, indirectly accessed by a stack pointer, that shrinks and
expands dynamically, and holds local function data. Items in the stack are
removed on a LIFO (last-in, first-out) basis.

Token
>   Is a fundamental symbol that represents a name or entity in a programming
>   language.

Thumb, Thumb2
>   An instruction set for ARM and Cortex devices. See **instruction set**.

UART
>   Universal Asynchronous Receiver/Transmitter
>   Is an individual IC, or part of an IC, used for serial communications.

Index