KEIL
SOFTWARE

Getting Started and Creating Applications

with µVision2 and the C51 Microcontroller Development Tools

User's Guide 09.99

NOTE
This manual assumes that you are familiar with Microsoft Windows and the hardware and instruction set of the 8051 microcontrollers.

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 Software development tools for the Infineon Technologies (formerly Siemens) 51 and ST Microelectronics ST10 family of microcontrollers. It introduces new users and interested readers to our products. This user's guide contains the following chapters.

"Chapter 1. Introduction" gives an overview and discusses the different products that Keil Software offers for the 8051 microcontroller families.

"Chapter 2. Installation" describes how to install the software and how to setup the operating environment for the tools.

"Chapter 3. Development Tools" describes the major features of the µVision2 IDE with integrated debugger, the C compiler, assembler, and utilities.

"Chapter 4. Creating Applications" describes how to create projects, edit source files, compile and fix syntax errors, and generate executable code.

"Chapter 5. Testing Programs" describes how you use the µVision2 debugger to simulate and test your entire application.

"Chapter 6. µVision2 Debug Functions" discusses built-in, user, and signal functions that extended the debuging capabilities of µVision2.

"Chapter 7. Sample Programs" provides several sample programs that show you how to use the Keil 8051 development tools.

"Chapter 8. RTX-51 Real-Time Operating System" discusses RTX-51 Tiny and RTX-51 Full and provides an example program.

"Chapter 9. Using on-chip Peripherals" shows how to access the on-chip 8051 peripherals with the C51 compiler. This chapter also includes several Application Notes.

"Chapter 10. CPU and C Startup Code" provides information on setting up the 8051 CPU for your application.

"Chapter 11. Using Monitor-51" discusses how to initialize the monitor and install it on your target hardware.

"Chapter 12. Command Reference" briefly describes the commands and controls available in the Keil 8051 development tools.

Document Conventions

This document uses the following conventions:

Examples	Description
README.TXT	Bold capital text is used for the names of executable programs, data files, source files, environment variables, and commands you enter at the command prompt. This text usually represents commands that you must type in literally. For example: **CLS** **DIR** **L51.EXE** Note that you are not required to enter these commands using all capital letters.
`Courier`	Text in this typeface is used to represent information that displays on screen or prints at the printer. This typeface is also used within the text when discussing or describing command line items.
Variables	
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 is omitted. For example:
	`Void main (void) {`
	`:`
	`:`
	`while (1);`

⟦*Optional Items*⟧	Optional arguments in command lines and optional fields are indicated by double brackets. For example: **`C51 TEST.C PRINT`** ⟦**`(`*`filename`*`)`**⟧
{ *opt1* | *opt2* }	Text contained within braces, separated by a vertical bar represents a group of items from which one must be chosen. The braces enclose all of the choices and the vertical bars separate the choices. 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.”
Point	Move the mouse until the mouse pointer rests on the item desired.
Click	Quickly press and release a mouse button while pointing at the item to be selected.
Drag	Press the left mouse button while on a selected item. Then, hold the button down while moving the mouse. When the item to be selected is at the desired position, release the button.
Double-Click	Click the mouse button twice in rapid succession.

Contents

Chapter 1. Introduction

Thank you for allowing Keil Software to provide you with software development tools for the 8051 family of microprocessors. With the Keil tools, you can generate embedded applications for multitude of 8051 derivatives.

NOTE
*Throughout this manual we refer to these tools as the **8051** development tools. However, they support all derivatives and variants of the 8051 microcontroller family.*

The Keil Software 8051 development tools listed below are the programs you use to compile your C code, assemble your assembler source files, link your program together, create HEX files, and debug your target program. Each of these programs is described in more detail in “Chapter 3. Development Tools” on page 15.

- µVision2 for Windows™ Integrated Development Environment: combines Project Management, Source Code Editing, and Program Debugging in one powerful environment.
- C51 ANSI Optimizing C Cross Compiler: creates relocatable object modules from your C source code,
- A51 Macro Assembler: creates relocatable object modules from your 8051 assembler source code,
- BL51 Linker/Locator: combines relocatable object modules created by the compiler and assembler into the final absolute object module,
- LIB51 Library Manager: combines object modules into a library which may be used by the linker,
- OH51 Object-HEX Converter: creates Intel HEX files from absolute object modules,
- RTX-51 real-time operating system: simplifies the design of complex, time-critical software projects.

The tools are combined into the kits described in “Product Overview” on page 8. They are designed for the professional software developer, but any level of programmer can use them to get the most out of the 8051 hardware.

Manual Topics

This manual discusses a number of topics including how to:

- Select the best tool kit for your application (see "Product Overview" on page 8),
- Install the software on your system (see "Chapter 2. Installation" on page 11),
- Overview and features of the 8051 development tools (see "Chapter 3. Development Tools" on page 15),
- Create full applications using the µVision2 integrated development environment (see "Chapter 4. Creating Applications" on page 47),
- Debug programs and simulate target hardware using the µVision2 debugger (see "Chapter 5. Testing Programs" on page 81),
- Access the on-chip peripherals and special features of the 8051 variants using the C51 compiler (see "Chapter 8. RTX-51 Real-Time Operating System
- " on page 157),
- Run the included sample programs (see "Chapter 7. Sample Programs" on page 141).

NOTE
If you want to get started immediately, you may do so by installing the software (refer to "Chapter 2. Installation" on page 11) and running the sample programs (refer to "Chapter 7. Sample Programs" on page 141).

Changes to the Documentation

Last minute changes and corrections to the software and manuals are listed in the **RELEASE.TXT** files. These files are located in the folders **UV2** and **C51\HLP**. Take the time to read this file to determine if there are any changes that may impact your installation.

Evaluation Kits and Production Kits

Keil Software provides two types of kits in which our tools are delivered.

The **EK51 Evaluation Kit** includes evaluation versions of our 8051 tools along with this user's guide. The tools in the evaluation kit let you generate applications up to 2 Kbytes in size. You may use this kit to evaluate the effectiveness of our 8051 tools and to generate small target applications.

The **8051 Production Kits** (discussed in "Product Overview" on page 8) include the unlimited versions of our 8051 tools along with this user's guide and the full manual set. The production kits also include 1 year of free technical support and product updates. Updates are available on world wide web ***www.keil.com*** under the update section.

Types of Users

This manual addresses three types of users: evaluation users, new users, and experienced users.

Evaluation Users are those users who have not yet purchased the software but have requested the evaluation package to get a better feel for what the tools do and how they perform. The evaluation package includes evaluation tools that are limited to 2 Kbytes along with several sample programs that provide real-world applications created for the 8051 microcontroller family. Even if you are only an evaluation user, take the time to read this manual. It explains how to install the software, provides you with an overview of the development tools, and introduces the sample programs.

New Users are those users who are purchasing 8051 development tools for the first time. The included software provides you with the latest development tool technology, manuals, and sample programs. If you are new to the 8051 or the tools, take the time to review the sample programs described in this manual. They provide a quick tutorial and help new or inexperienced users quickly get started.

Experienced Users are those users who have previously used the Keil 8051 development tools and are now upgrading to the latest version. The software included with a product upgrade contains the latest development tools and sample programs.

Requesting Assistance

At Keil Software, we are dedicated to providing you with the best embedded development tools and documentation available. If you have suggestions or comments regarding any of the printed manuals accompanying this product, please contact us. If you think you have discovered a problem with the software, do the following before calling technical support.

1. Read the sections in this manual that pertains to the job or task you are trying to accomplish.
2. Make sure you are using the most current version of the software and utilities. Check out the update section on ***www.keil.com*** to make sure that you have the latest software version.
3. Isolate the problem to determine if it is a problem with the assembler, compiler, linker, library manager, or another development tool.
4. Further isolate software problems by reducing your code to a few lines.

If, after following these steps, you are still experiencing problems, report them to our technical support group. Please include your product serial number and version number. We prefer that you send the problem via email. If you contact us by fax, be sure to include your name and telephone numbers (voice and fax) where we can reach you.

Try to be as detailed as possible when describing the problem you are having. The more descriptive your example, the faster we can find a solution. If you have a one-page code example demonstrating the problem, please email it to us. If possible, make sure that your problem can be duplicated with the µVision2 simulator. Please try to avoid sending complete applications or long listings as this slows down our response to you.

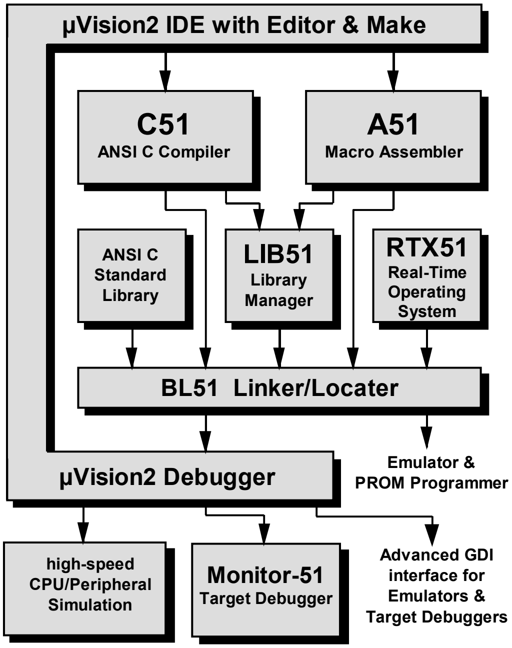
NOTE
You can always get technical support, product updates, application notes, and sample programs from our world wide web site **www.keil.com.**

Software Development Cycle

When you use the Keil Software tools, the project development cycle is roughly the same as it is for any other software development project.

1. Create a project to select the 8051 device and the tool settings.
2. Create source files in C or assembly.
3. Build your application with the project manager.
4. Correct errors in source files.
5. Test linked application.

The development cycle described above may be best illustrated by a block diagram of the complete 8051 tool set.


µVision2 IDE with Editor & Make
C51
ANSI C Compiler
A51
Macro Assembler
ANSI C Standard Library
LIB51
Library Manager
RTX51
Real-Time Operating System
BL51 Linker/Locater
µVision2 Debugger
Emulator & PROM Programmer
high-speed CPU/Peripheral Simulation
Monitor-51
Target Debugger
Advanced GDI interface for Emulators & Target Debuggers

µVision2 IDE

The µVision2 IDE combines project management, a rich-featured editor with interactive error correction, option setup, make facility, and on-line help.

You use µVision2 to create your source files and organize them into a project that defines your target application. µVision2 automatically compiles, assembles, and links your embedded application and provides a single focal point for you development efforts.

8051 Compiler & Assembler

Source files are created by the µVision2 IDE and are passed to the C51 compiler or A51 assembler. The compiler and assembler process source files and creates relocatable object files. The Keil C51 compiler is a full ANSI implementation of the C programming language. All standard features of the C language are supported. In addition, numerous features for direct support of the 8051 environment have been added. The Keil A51 macro assembler supports the complete instruction sets of the 8051 and all derivatives.

LIB51 Library Manager

Object files created by the compiler and assembler may be used by the LIB51 library manager to create object libraries which are specially formatted, ordered program collections of object modules that the linker may process at a later time. When the linker processes a library, only those object modules in the library that are necessary to create the program are used.

BL51 Linker/Locator

Object files and library files are processed by the linker into an absolute object module. An absolute object file or module contains no relocatable code. All the code in an absolute object file resides at fixed memory locations. The absolute object file may be used to program EPROM or other memory devices. The absolute object module may also be used with the µVision2 Debugger or with an in-circuit emulator for the program test.

µVision2 Debugger

The µVision2 symbolic, source-level debugger is ideally suited for fast, reliable program debugging. The debugger contains a high-speed simulator that let you simulate an entire 8051 system including on-chip peripherals and external hardware. Via the integrated device database you can configure the µVision2 debugger to the attributes and peripherals of 8051 device you are using.

For testing the software in a real hardware, you may connect the µVision2 Debugger with Monitor-51 or you can use the *Advanced GDI* interface to attach the debugger front-end to a target system.

Monitor-51

The µVision2 Debugger supports target debugging using Monitor-51. The monitor program is a program that resides in the memory of your target hardware and communicates with µVision2 using the serial port of the 8051 and a COM port of your PC. With Monitor-51, µVision2 lets you perform source-level, symbolic debugging on your target hardware.

RTX51 Real-Time Operating System

The RTX51 real-time operating system is a multitasking kernel for the 8051 family. The RTX51 real-time kernel simplifies the system design, programming, and debugging of complex applications where fast reaction to time critical events is essential. The kernel is fully integrated into the C51 compiler and is easy to use. Task description tables and operating system consistency are automatically controlled by the BL51 linker/locator.

Product Overview

Keil Software provides the premier development tools for the 8051 family of microcontrollers. We bundle our software development tools into different packages or tool kits. The "Comparison Chart" on page 9 shows the full extent of the Keil Software 8051 development tools. Each kit and its contents are described below.

PK51 Professional Developer's Kit

The **PK51** Professional Developer's Kit includes everything the professional developer needs to create and debug sophisticated embedded applications for the 8051 family of microcontrollers. The professional developer's kit can be configured for all 8051 derivatives.

DK51 Developer's Kit

The **DK51** Developer's Kit is a reduced version of PK51 and does not include the RTX51 Tiny real-time operating system. The developer's kit can be configured for all 8051 derivatives.

CA51 Compiler Kit

The CA51 Compiler Kit is the best choice for developers who need a C compiler but not a debugging system. The CA51 package contains only the µVision IDE. The µVision2 Debugger features are not available in CA51. The kit includes everything you need to create embedded applications and can be configured for all 8051 derivatives.

A51 Assembler Kit

The A51 Assembler Kit includes an assembler and all the utilities you need to create embedded applications. It can be configured for all 8051 derivatives.

RTX51 Real-Time Operating System (FR51)

The RTX51 Real-Time Operating Systems is a real-time kernel for the 8051 family of microcontrollers. RTX51 Full provides a superset of the features found in RTX51 Tiny and includes CAN communication protocol interface routines.

Comparison Chart

The following table provides a check list of the features found in each package. Tools are listed along the top and part numbers for specific kits are listed along the side. Use this cross reference to select the kit that best suits your needs.

Compontents	PK51	DK51†	CA51	A51	FR51
µVision2 Project Management & Editor					
A51 Assembler					
C51 Compiler					
BL51 Linker/Locator					
LIB51 Library Manager					
µVision2 Debugger/Simulator					
RTX51 Tiny					
RTX51 Full					

Chapter 2. Installation

This chapter explains how to setup an operating environment and how to install the software on your hard disk. Before starting the installation program, you must do the following:

- Verify that your computer system meets the minimum requirements.
- Make a copy of the installation diskette for backup purposes.

System Requirements

There are minimum hardware and software requirements that must be satisfied to ensure that the compiler and utilities function properly.

For our Windows-based tools, you must have the following:

- PC with Pentium, Pentium-II or compatible processor,
- Windows 95, Windows-98, Windows NT 4.0, or higher
- 16 MB RAM minimum,
- 20 MB free disk space.

Installation Details

All of our products come with an installation program that allows easy installation of our software. To install the 8051 development tools:

- Insert the Keil Development Tools CD-ROM.
- Select **Install Software** from the Keil CD Viewer menu and follow the instructions displayed by the setup program.

NOTE
*Your PC should automatically launch the CD Viewer when you insert the CD. If not, run the program **KEIL\SETUP\SETUP.EXE** from the CD to install the software.*

Folder Structure

The setup program copies the development tools into sub-folders of the base folder. The default base folder is: **C:\KEIL**. The following table lists the structure of a complete installation that includes the entire line of 8051 development tools. Your installation may vary depending on the products you purchased.

Folder	Description
C:\KEIL\C51\ASM	Assembler SFR definition files and template source file.
C:\KEIL\C51\BIN	Executable files of the 8051 toolchain.
C:\KEIL\C51\EXAMPLES	Sample applications.
C:\KEIL\C51\RTX51	RTX51 Full files.
C:\KEIL\C51\RTX_TINY	RTX51 Tiny files.
C:\KEIL\C51\INC	C compiler include files.
C:\KEIL\C51\LIB	C compiler library files, startup code, and source of I/O routines.
C:\KEIL\C51\MONITOR	Target Monitor files and Monitor configuration for user hardware.
C:\KEIL\UV2	Generic µVision2 files.

Within this users guide we refer to the default folder structure. If you have installed your software on a different folder, you have to adjust the pathnames to match with your installation.

Chapter 3. Development Tools

This chapter discusses the features and advantages of the 8051 development tools available from Keil Software. We have designed our tools to help you quickly and successfully complete your job. They are easy to use and are guaranteed to help you achieve your design goals.

µVision2 Integrated Development Environment

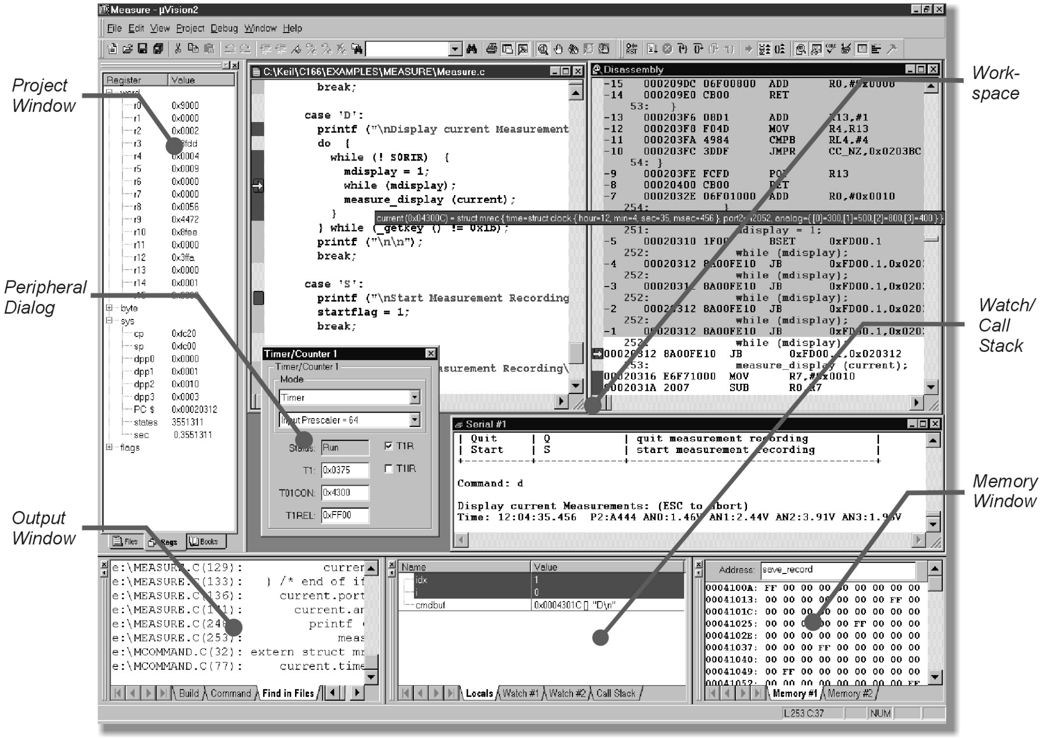
µVision2 is a standard Windows application. µVision2 is an integrated software development platform that combines a robust editor, project manager, and make facility. µVision2 supports all of the Keil tools for the 8051 including the C compiler, macro assembler, linker/locator, and object-HEX converter. µVision2 helps expedite the development process of your embedded applications by providing the following:

- Full-featured source code editor,
- Device Database for pre-configuring the development tool setting,
- Project manager for creating and maintaining your projects,
- Integrated make facility for assembling, compiling, and linking your embedded applications,
- Dialogs for all development tool settings,
- True integrated source-level Debugger with high-speed CPU and peripheral simulator.
- Advanced GDI interface for software debugging in the target hardware and for connection to Monitor-51.
- Links to development tools manuals, device datasheets & user’s guides.

NOTE
The µVision2 debugging features are only available in the ***PK51*** *and* ***DK51*** *tool kits.*

About the Environment

The µVision2 screen provides you with a menu bar for command entry, a tool bar where you can rapidly select command buttons, and windows for source files, dialog boxes, and information displays. µVision2 lets you simultaneously open and view multiple source files.



```
Register    Value
⊟ word
    r0      0x9000
    r1      0x0000
    r2      0x0002
    r3      0x3fdd
    r4      0x0004
    r5      0x0009
    r6      0x0000
    r7      0x0000
    r8      0x0056
    r9      0x4472
    r10     0x8fee
    r11     0x0000
    r12     0x3ffa
    r13     0x0000
    r14     0x0001
    r15     0x0000
⊞ byte
⊟ sys
    cp      0xfc20
    sp      0xfc00
    dpp0    0x0000
    dpp1    0x0001
    dpp2    0x0010
    dpp3    0x0003
    PC $    0x00020312
    states  3551311
    sec      0.3551311
⊞ flags
```

C:\Keil\C166\EXAMPLES\MEASURE\Measure.c

```
        break;

      case 'D':
        printf ("\nDisplay current Measurement
        do  {
          while (! S0RIR)  {
            mdisplay = 1;
            while (mdisplay);
            measure_display (current);
          }
        } while (_getkey () != 0x1b);
        printf ("\n\n");
        break;

      case 'S':
        printf ("\nStart Measurement Recording
        startflag = 1;
        break;

                         asurement Recording\
```

current (0x04300C) = struct mrec { time=struct clock { hour=12, min=4, sec=35, msec=456 }, port2=42052, analog={ [0]=300,[1]=500,[2]=800,[3]=400 } }

Timer/Counter 1

Timer/Counter 1

Mode

Timer

Input Prescaler = 64

Status: Run ☑ T1R

T1: 0x0375 ☐ T1IR

T01CON: 0x4300

T1REL: 0xFF00

Disassembly

```
-15    000209DC 06F00800   ADD        R0,#0x0008
-14    000209E0 CB00       RET
   53:    }
-13    000203F6 08D1       ADD        R13,#1
-12    000203F8 F04D       MOV        R4,R13
-11    000203FA 4984       CMPB       RL4,#4
-10    000203FC 3DDF       JMPR       CC_NZ,0x0203BC
   54: }
-9     000203FE FCFD       POP        R13
-8     00020400 CB00       RET
-7     0002032E 06F01000   ADD        R0,#0x0010
  254:                 }
  251:                 mdisplay = 1;
-5     00020310 1F00       BSET       0xFD00.1
  252:                 while (mdisplay);
-4     00020312 8A00FE10   JB         0xFD00.1,0x020:
  252:                 while (mdisplay);
-3     00020312 8A00FE10   JB         0xFD00.1,0x020:
  252:                 while (mdisplay);
-2     00020312 8A00FE10   JB         0xFD00.1,0x020:
  252:                 while (mdisplay);
-1     00020312 8A00FE10   JB         0xFD00.1,0x020:
  252:                 while (mdisplay);
00020312 8A00FE10   JB         0xFD00.1,0x020312
  253:                 measure_display (current);
00020316 E6F71000   MOV        R7,#0x0010
0002031A 2007       SUB        R0,R7
```

Serial #1

```
| Quit       | Q           | quit measurement recording         |
| Start      | S           | start measurement recording        |
+------------+-------------+------------------------------------+

Command: d

Display current Measurements: (ESC to abort)
Time: 12:04:35.456  P2:A444 AN0:1.46V AN1:2.44V AN2:3.91V AN3:1.95V
```

```
e:\MEASURE.C(129):                curren
e:\MEASURE.C(133):    } /* end of if
e:\MEASURE.C(136):       current.port
e:\MEASURE.C(141):         current.an
e:\MEASURE.C(248):           printf (
e:\MEASURE.C(253):                meas
e:\MCOMMAND.C(32): extern struct mr
e:\MCOMMAND.C(77):       current.time
```

Name	Value
idx	1
i	0
cmdbuf	0x0004301C ▯ "D\n"

Address: save_record

```
0004100A: FF 00 00 00 00 00 00 00 00
00041013: 00 00 00 00 00 00 00 FF 00
0004101C: 00 00 00 00 00 00 00 00 00
00041025: 00 00 00 00 00 FF 00 00 00
0004102E: 00 00 00 00 00 00 00 00 00
00041037: 00 00 00 FF 00 00 00 00 00
00041040: 00 00 00 00 00 00 00 00 00
00041049: 00 FF 00 00 00 00 00 00 00
00041052: 00 00 00 00 00 00 00 00 FF
```

Menu Commands, Toolbars and Shortcuts

The menu bar provides you with menus for editor operations, project maintenance, development tool option settings, program debugging, window selection and manipulation, and on-line help. With the toolbar buttons you can rapidly execute operations. The commands can be reached also with configurable keyboard shortcuts. The following tables give you an overview of the µVision2 commands and the default shortcuts.

File Menu and File Commands

Toolbar	File Menu	Shortcut	Description
	New	Ctrl+N	Create a new source or text file
	Open	Ctrl+O	Open an existing file
	Close		Close the active file
	Save	Ctrl+S	Create a new source or text file
			Save all open source and text files
	Save as…		Save and rename the active file
	Device Database		Maintain the µVision2 device database
	Print Setup…		Setup the printer
	Print	Ctrl+P	Print the active file
	Print Preview		Display pages in print view
	1 .. 9		Open the most recent used source or text files
	Exit		Quit µVision2 and prompt for saving files

Edit Menu and Editor Commands

Toolbar	Edit Menu	Shortcut	Description
		Home	Move cursor to beginning of line
		End	Move cursor to end of line
		Ctrl+Home	Move cursor to beginning of file
		Ctrl+End	Move cursor to end of file
		Ctrl+	Move cursor one word left
		Ctrl+	Move cursor one word rigth
		Ctrl+A	Select all text in the current file
	Undo	Ctrl+Z	Undo last operation
	Redo	Ctrl+Shift+Z	Redo last undo command
	Cut	Ctrl+X	Cut selected text to clipboard
		Ctrl+Y	Cut text in the current line to clipboard
	Copy	Ctrl+C	Copy selected text to clipboard
	Paste	Ctrl+V	Paste text from clipboard
	Indent Selected Text		Indent selected text right one tab stop
	Unindent Selected Text		Indent selected text left one tab stop
	Toggle Bookmark	Ctrl+F2	Toggle bookmark at current line
	Goto Next Bookmark	F2	Move cursor to next bookmark
	Goto Previous Bookmark	Shift+F2	Move cursor to previous bookmark
	Clear All Bookmarks		Clear all bookmarks in active file
command ▾	Find	Ctrl+F	Search text in the active file
		F3	Repeat search text forward

Toolbar	Edit Menu	Shortcut	Description
		Shift+F3	Repeat search text backward
		Ctrl+F3	Search word under cursor
		Ctrl+]	Find matching brace, parenthesis, or bracket (to use this command place cursor before a brace, parenthesis or bracket)
	Replace	Ctrl+H	Replace specific text
	Find in Files…		Search text in several files

Select Text Commands

In µVision2 you can select text by holding down **Shift** and pressing the key that moves the cursor. For example, **Ctrl+** moves the cursor to the next word, and **Ctrl+Shift+** selects the text from the current cursor position to the beginning of the next word. With the mouse you can select text as follows:

Select Text	With the Mouse
Any amount of text	Drag over the text
A word	Double-click the word
A line of text	Move the pointer to the left of the line until it changes to a right-pointing arrow, and then click
Multiple lines of text	Move the pointer to the left of the lines until it changes to a right-pointing arrow, and then drag up or down
A vertical block of text	Hold down the ALT key, and then drag

View Menu

Toolbar	View Menu	Shortcut	Description
	Status Bar		Show or hide the status bar
	File Toolbar		Show or hide the File toolbar
	Build Toolbar		Show or hide the Build toolbar
	Debug Toolbar		Show or hide the Debug toolbar
	Project Window		Show or hide the Project window
	Output Window		Show or hide the Output window
	Source Browser		Open the Source Browser window
	Disassembly Window		Show or hide the Disassembly window
	Watch & Call Stack Window		Show or hide the Watch & Call Stack window
	Memory Window		Show or hide the Memory window
CODE ✓	Code Coverage Window		Show or hide the Code Coverage window
	Performance Analyzer Window		Show or hide the Performance Analyzer window
	Symbol Window		Show or hide the Symbol window
	Serial Window #1		Show or hide the Serial window #1

Toolbar	View Menu	Shortcut	Description
	Serial Window #2		Show or hide the Serial window #2
	Toolbox		Show or hide the Toolbox
	Periodic Window Update		Updates debug windows while running the program
	Workbook Mode		Show workbook frame with windows tabs
	Options…		Change Colors, Fonts, Shortcuts and Editor options

Project Menu and Project Commands

Toolbar	Project Menu	Shortcut	Description
	New Project …		Create a new project
	Import µVision1 Project …		Convert a µVision1 Project File (see page 70)
	Open Project …		Open an existing project
	Close Project…		Close current project
	Target Environment		Define paths for tool chain, include & library files
	Targets, Groups, Files		Maintain Targets, File Groups and Files of a project
	Select Device for Target		Select a CPU from the Device Database
	Remove…		Remove a Group or File from the project
	Options…	Alt+F7	Change tool options for Target, Group or File
			Change options for current Target
MCB251 ▾			Select current Target
	File Extensions		Select file extensions for different file types
	Build Target	F7	Translate modified files and build application
	Rebuild Target		Re-translate all source files and build application
	Translate…	Ctrl+F7	Translate current file
	Stop Build		Stop current build process
	1 .. 9		Open the most recent used project files

Debug Menu and Debug Commands

Toolbar	Debug Menu	Shortcut	Description
	Start/Stop Debugging	Ctrl+F5	Start or stop µVision2 Debug Mode
	Go	F5	Run (execute) until the next active breakpoint
	Step	F11	Execute a single-step into a function
	Step over	F10	Execute a single-step over a function
	Step out of current function	Ctrl+F11	Execute a step out of the current function
✖	Stop Running	ESC	Stop program execution
	Breakpoints…		Open Breakpoint dialog
	Insert/Remove Breakpoint		Toggle breakpoint on current line
	Enable/Disable Breakpoint		Enable/disable breakpoint on the current line

Toolbar	Debug Menu	Shortcut	Description
✋	Disable All Breakpoints		Disable all breakpoints in the program
✋	Kill All Breakpoints		Kill all breakpoints in the program
⇨	Show Next Statement		Show next executeable statement/instruction
REC	Enable/Disable Trace Recording		Enable trace recording for instruction review
{}	View Trace Records		Review previous executed instructions
	Memory Map…		Open memory map dialog
	Performance Analyzer…		Open setup dialog for the Performance Analyzer
	Inline Assembly…		Stop current build process
	Function Editor…		Edit debug functions and debug INI file

Peripherals Menu

Toolbar	Peripherals Menu	Shortcut	Description
RST	Reset CPU		Set CPU to reset state
	[Interrupt … Watchdog]		Open dialogs for on-chip peripherals, these dialogs depend on the CPU selected from the device database

Tools Menu

The tools menu allows you to configure and run Gimpel PC-Lint, Siemens Easy-Case, and custom programs. With **Customize Tools Menu…** user programs are added to the menu. For more information refer to “Using the Tools Menu” on page 61.

Toolbar	Tools Menu	Shortcut	Description
	Setup PC-Lint…		Configure PC-Lint from Gimpel Software
	Lint		Run PC-Lint current editor file
	Lint all C Source Files		Run PC-Lint across the C source files of your project
	Setup Easy-Case…		Configure Siemens Easy-Case
	Start/Stop Easy-Case		Start or stop Siemens Easy-Case
	Show File (Line)		Open Easy-Case with the current editor file
	Customize Tools Menu…		Add user programs to the Tools Menu

SVCS Menu

With the SVCS menu you configure and add the commands of a Software Version Control System (SVCS). For more information refer to “Using the SVCS Menu” on page 64..

Toolbar	SVCS Menu	Shortcut	Description
	Configure Version Control…		Configure the commands of your SVCS

Window Menu

Toolbar	Window Menu	Shortcut	Description
	Cascade		Arrange Windows so they overlap
	Tile Horizontally		Arrange Windows so they no overlap
	Tile Vertically		Arrange Windows so they no overlap
	Arrange Icons		Arrange Icons at the bottom of the window
	Split		Split the active window into panes
	1 .. 9		Activate the selected window

Help Menu

Toolbar	Help Menu	Shortcut	Description
	Help topics		Open on-line help
	About µVision		Display version numbers and license information

µVision2 has two operating modes:

- **Build Mode:** allows you to translate all the application files and to generate executable programs. The features of the Build Mode are described in ”Chapter 4. Creating Applications” on page 47.
- **Debug Mode:** provides you with a powerful debugger for testing your application. The Debug Mode is described in “Chapter 5. Testing Programs” on page 81.

In both operating modes you can use the source editor of µVision2 to modify your source code.

C51 Optimizing C Cross Compiler

For 8051 microcontroller applications, the Keil C51 Cross Compiler offers a way to program in C which truly matches assembly programming in terms of code efficiency and speed. The Keil C51 is not a universal C compiler adapted for the 8051. It is a dedicated C compiler that generates extremely fast and compact code. The Keil C51 Compiler implements the ANSI standard for the C language.

Use of a high-level language such as C has many advantages over assembly language programming:

- Knowledge of the processor instruction set is not required, rudimentary knowledge of the memory structure of the 8051 CPU is desirable (but not necessary).
- Details like register allocation and addressing of the various memory types and data types is managed by the compiler.
- Programs get a formal structure and can be divided into separate functions. This leads to better program structure.
- The ability to combine variable selection with specific operations improves program readability.
- Keywords and operational functions can be used that more nearly resemble the human thought process.
- Programming and program test time is drastically reduced which increases your efficiency.
- The C run-time library contains many standard routines such as: formatted output, numeric conversions and floating point arithmetic.
- Existing program parts can be more easily included into new programs, because of the comfortable modular program construction techniques.
- The language C is a very portable language (based on the ANSI standard) that enjoys wide popular support, and can be easily obtained for most systems. This means that existing program investments can be quickly adapted to other processors as needed.

C51 Language Extensions

The C51 compiler is an ANSI compliant C compiler and includes all aspects of the C programming language that are specified by the ANSI standard. A number of extensions to the C programming language are provided to support the facilities of the 8051 microprocessor. The C51 compiler includes extensions for:

- Data Types,
- Memory Types,
- Memory Models,
- Pointers,
- Reentrant Functions,
- Interrupt Functions,
- Real-Time Operating Systems,
- Interfacing to PL/M and A51 source files.

The following sections briefly describe these extensions.

Data Types

The C51 compiler supports the data types listed in the following table. In addition to these scalar types, variables can be combined into structures, unions, and arrays. Except as noted, you may use pointers to access these data types.

Data Type	Bits	Bytes	Value Range
bit †	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to 2147483647
unsigned long	32	4	0 to 4294967295
float	32	4	±1.175494E-38 to ±3.402823E+38
sbit †	1		0 to 1
sfr †	8	1	0 to 255
sfr16 †	16	2	0 to 65535

† The **bit**, **sbit**, **sfr**, and **sfr16** data types are specific to the 8051 hardware and the C51 and C251 compilers. The are not a part of ANSI C and cannot be accessed through pointers.

The **sbit**, **sfr**, and **sfr16** data types are included to allow access to the special function registers that are available on the 8051. For example, the declaration: `sfr P0 = 0x80;` declares the variable `P0` and assigns it the special function register address of `0x80`. This is the address of PORT 0 on the 8051.

The C51 compiler automatically converts between data types when the result implies a different data type. For example, a bit variable used in an integer assignment is converted to an integer. You can, of course, coerce a conversion by using a type cast. In addition to data type conversions, sign extensions are automatically carried out for signed variables.

Memory Types

The C51 compiler supports the architecture of the 8051 and its derivatives and provides access to all memory areas of the 8051. Each variable may be explicitly assigned to a specific memory space.

Memory Type	Description
code	Program memory (64 Kbytes); accessed by opcode **MOVC @A+DPTR**.
data	Directly addressable internal data memory; fastest access (128 bytes).
idata	Indirectly addressable internal data memory; accessed across the full internal address space (256 bytes).
bdata	Bit-addressable internal data memory; mixed bit and byte access (16 bytes).
xdata	External data memory (64 Kbytes); accessed by opcode **MOVX @DPTR**.
pdata	Paged (256 bytes) external data memory; accessed by opcode **MOVX @Rn**.

Accessing the internal data memory is considerably faster than accessing the external data memory. For this reason, you should place frequently used variables in internal data memory and less frequently used variables in external data memory.

By including a memory type specifier in the variable declaration, you can specify where variables are stored.

As with the **signed** and **unsigned** attributes, you may include memory type specifiers in the variable declaration. For example:

```
char data var1;
char code text[] = "ENTER PARAMETER:";
unsigned long xdata array[100];
float idata x,y,z;
unsigned int pdata dimension;
unsigned char xdata vector[10][4][4];
char bdata flags;
```

If the memory type specifier is omitted in a variable declaration, the default or implicit memory type is automatically selected. Function arguments and automatic variables which cannot be located in registers are also stored in the default memory area.

The default memory type is determined by the **SMALL**, **COMPACT** and **LARGE** compiler control directives. These directives specify the memory model to use for the compilation.

Memory Models

The memory model determines the default memory type used for function arguments, automatic variables, and variables declared with no explicit memory type. You specify the memory model on the command line using the **SMALL**, **COMPACT**, and **LARGE** control directives. By explicitly declaring a variable with a memory type specifier, you may override the default memory type.

SMALL All variables default to the internal data memory of the 8051. This is the same as if they were declared explicitly using the **data** memory type specifier. In this memory model, variable access is very efficient. However, all data objects, as well as the stack must fit into the internal RAM. Stack size is critical because the stack space used depends upon the nesting depth of the various functions. Typically, if the BL51 code banking linker/locator is configured to overlay variables in the internal data memory, the small model is the best model to use.

COMPACT All variables default to one page of external data memory. This is the same as if they were explicitly declared using the **pdata** memory type specifier. This memory model can accommodate a maximum of 256 bytes of variables. The limitation is due to the addressing scheme used, which is indirect through registers R0 and R1. This memory model is not as efficient as the small model, therefore, variable access is not as fast. However, the compact model is faster than the large model. The high byte of the address is usually set up via port 2. The compiler does not set this port for you.

LARGE In large model, all variables default to external data memory. This is the same as if they were explicitly declared using the **xdata** memory type specifier. The data pointer (**DPTR**) is used for addressing. Memory access through this data pointer is inefficient, especially for variables with a length of two or more bytes. This type of data access generates more code than the small or compact models.

NOTE
You should always use the ***SMALL*** *memory model. It generates the fastest, tightest, and most efficient code. You can always explicitly specify the memory area for variables. Move up in model size only if you are unable to make your application fit or operate using* ***SMALL*** *model.*

Pointers

The C51 compiler supports pointer declarations using the asterisk character (‘*’). You may use pointers to perform all operations available in standard C. However, because of the unique architecture of the 8051 and its derivatives, the C51 compiler supports two different types of pointers: memory specific pointers and generic pointers.

Generic Pointers

Generic pointers are declared in the same way as standard C pointers. For example:

```
char *s;                                                   /* string ptr */
int *numptr;                                                  /* int ptr */
long *state;                                                 /* long ptr */
```

Generic pointers are always stored using three bytes. The first byte is for the memory type, the second is for the high-order byte of the offset, and the third is for the low-order byte of the offset.

Generic pointers may be used to access any variable regardless of its location in 8051 memory space. Many of the library routines use these pointer types for this reason. By using these generic untyped pointers, a function can access data regardless of the memory in which it is stored.

Memory Specific Pointers

Memory specific pointers always include a memory type specification in the pointer declaration and always refer to a specific memory area. For example:

```
char data *str;                                 /* ptr to string in data */
int xdata *numtab;                             /* ptr to int(s) in xdata */
long code *powtab;                             /* ptr to long(s) in code */
```

Because the memory type is specified at compile-time, the memory type byte required by untyped pointers is not needed by typed pointers. Typed pointers can be stored using only one byte (**idata**, **data**, **bdata**, and **pdata** pointers) or two bytes (**code** and **xdata** pointers).

Comparison: Memory Specific & Generic Pointers

You can significantly accelerate an 8051 C program by using ‘memory specific’ pointers. The following sample program shows the differences in code & data size and execution time for various pointer declarations.

Description	Idata Pointer	Xdata Pointer	Generic Pointer

Sample Program	`char idata *ip;`
	`char val;`
	`val = *ip;`
		`char xdata *xp;`
		`char val;`
		`val = *xp;`
			`char *p;`
			`char val;`
			`val = *p;`

8051 Program Code Generated	`MOV  R0,ip`
	`MOV  val,@R0`
		`MOV  DPL,xp +1`
		`MOV  DPH,xp`
		`MOV  A,@DPTR`
		`MOV  val,A`
			`MOV  R1,p + 2`
			`MOV  R2,p + 1`
			`MOV  R3,p`
			`CALL CLDPTR`

Pointer Size	1 byte data	2 bytes data	3 bytes data
Code Size	4 bytes code	9 bytes code	11 bytes code + Lib.
Execution Time	4 cycles	7 cycles	13 cycles

Reentrant Functions

A reentrant function can be shared by several processes at the same time. When a reentrant function is executing, another process can interrupt the execution and then begin to execute that same reentrant function. Normally, C51 functions cannot be called recursively or in a fashion which causes reentrancy. The reason for this limitation is that function arguments and local variables are stored in fixed memory locations. The **reentrant** function attribute allows you to declare functions that may be reentrant and, therefore, may be called recursively. For example:

```
int calc (char i, int b) reentrant
  {
  int  x;
  x = table [i];
  return (x * b);
  }
```

Reentrant functions can be called recursively and can be called *simultaneously* by two or more processes. Reentrant functions are often required in real-time applications or in situations where interrupt code and non-interrupt code must share a function.

For each reentrant function, a reentrant stack area is simulated in internal or external memory depending on the memory model.

NOTE
*By selecting the **reentrant** attribute on a function by function basis, you can select the use of this attribute where it's needed without making the entire program **reentrant**. Making an entire program reentrant may cause it to be larger and consume more memory.*

Interrupt Functions

The C51 compiler provides you with a method of calling a C function when an interrupt occurs. This support allows you to create interrupt service routines in C. You need only be concerned with the interrupt number and register bank selection. The compiler automatically generates the interrupt vector and entry and exit code for the interrupt routine. The **interrupt** function attribute, when included in a declaration, specifies that the associated function is an interrupt function. Additionally, you can specify the register bank used for that interrupt with the **using** function attribute.

```
unsigned int interruptcnt;
unsigned char second;

void timer0 (void) interrupt 1 using 2 {
  if (++interruptcnt == 4000) {                      /* count to 4000 */
    second++;                                        /* second counter */
    interruptcnt = 0;                              /* clear int counter */
  }
}
```

Parameter Passing

The C51 compiler passes up to three function arguments in CPU registers. This significantly improves system performance since arguments do not have to be written to and read from memory. Argument passing can be controlled with the **REGPARMS** and **NOREGPARMS** control directives. The following table lists the registers used for different arguments and data types.

Argument Number	char, 1-byte pointer	int, 2-byte pointer	long, float	generic pointer
1	R7	R6 & R7	R4 — R7	R1 — R3
2	R5	R4 & R5		
3	R3	R2 & R3		

If no registers are available for argument passing or too many arguments are involved, fixed memory locations are used for those extra arguments.

Function Return Values

CPU registers are always used for function return values. The following table lists the return types and the registers used for each.

Return Type	Register	Description
bit	Carry Flag	
char, **unsigned char, 1-byte pointer**	R7	
int, **unsigned int, 2-byte pointer**	R6 & R7	MSB in R6, LSB in R7
long, **unsigned long**	R4 — R7	MSB in R4, LSB in R7
float	R4 — R7	32-Bit IEEE format
generic pointer	R1 — R3	Memory type in R3, MSB R2, LSB R1

Register Optimizing

Depending on program context, the C51 compiler allocates up to 7 CPU registers for register variables. Any registers modified during function execution are noted by the C51 compiler within each module. The linker/locator generates a global, project-wide register file which contains information of all registers altered by external functions. Consequently, the C51 compiler *knows* the register used by each function in an application and can optimize the CPU register allocation of each C function.

Real-Time Operating System Support

The C51 compiler integrates well with both the RTX-51 Full and RTX-51 Tiny multitasking real-time operating systems. The task description tables are generated and controlled during the link process. For more information about the RTX real-time operating systems, refer to "**Error! Reference source not found.**" on page **Error! Bookmark not defined.**.

Interfacing to Assembly

You can easily access assembly routines from C and vice versa. Function parameters are passed via CPU registers or, if the **NOREGPARMS** control is used, via fixed memory locations. Values returned from functions are always passed in CPU registers.

You can use the **SRC** directive to direct the C51 compiler to generate a file ready to assemble with the A51 assembler instead of an object file. For example, the following C source file:

```
unsigned int asmfunc1 (unsigned int arg){
  return (1 + arg);
}
```

generates the following assembly output file when compiled using the **SRC** directive.

```
?PR?_asmfunc1?ASM1     SEGMENT CODE
PUBLIC                 _asmfunc1
                  RSEG  ?PR?_asmfunc1?ASM1
                  USING 0
_asmfunc1:
;---- Variable 'arg?00' assigned to Register 'R6/R7' ----
                  MOV   A,R7           ; load LSB of the int
                  ADD   A,#01H         ; add 1
                  MOV   R7,A           ; put it back into R7
                  CLR   A
                  ADDC  A,R6           ; add carry & R6
                  MOV   R6,A

?C0001:
                  RET                  ; return result in R6/R7
```

You may use the **#pragma asm** and **#pragma endasm** preprocessor directives to insert assembly instructions into your C source code.

Interfacing to PL/M-51

Intel's PL/M-51 is a popular programming language that is similar to C in many ways. You can easily interface routines written in C to routines written in PL/M-51. You can access PL/M-51 functions from C by declaring them with the **alien** function type specifier. All public variables declared in the PL/M-51 module are available to your C programs. For example:

```
extern alien char plm_func (int, char);
```

Since the PL/M-51 compiler and the Keil Software tools all generate object files in the OMF51 format, external symbols are resolved by the linker.

Code Optimizations

The C51 compiler is an aggressive optimizing compiler. This means that the compiler takes certain steps to ensure that the code generated and output to the object file is the most efficient (smaller and/or faster) code possible. The compiler analyzes the generated code to produce the most efficient instruction sequences. This ensures that your C program runs as quickly and effectively as possible in the least amount of code space.

The C51 compiler provides six different levels of optimizing. Each increasing level includes the optimizations of levels below it. The following is a list of all optimizations currently performed by the C51 compiler.

General Optimizations

- **Constant Folding**: Several constant values occurring in an expression or address calculation are combined as a single constant.
- **Jump Optimizing**: Jumps are inverted or extended to the final target address when the program efficiency is thereby increased.
- **Dead Code Elimination**: Code which cannot be reached (dead code) is removed from the program.
- **Register Variables**: Automatic variables and function arguments are located in registers whenever possible. No data memory space is reserved for these variables.
- **Parameter Passing Via Registers**: A maximum of three function arguments can be passed in registers.
- **Global Common Subexpression Elimination**: Identical subexpressions or address calculations that occur multiple times in a function are recognized and calculated only once whenever possible.
- **Common Tail Merging**: common instruction blocks are merged together using jump instructions.
- **Re-use Common Entry Code**: common instruction sequences are moved in front of a function to reduce code size
- **Common Block Subroutines**: multiple instruction sequences are packed into subroutines. Instructions are rearranged to maximize the block size.

8051-Specific Optimizations

- **Peephole Optimization**: Complex operations are replaced by simplified operations when memory space or execution time can be saved as a result.
- **Access Optimizing**: Constants and variables are computed and included directly in operations.
- **Extended Access Optimizing**: the DPTR register is used as register variable for memory specific pointers to improve code density.
- **Data Overlaying**: Data and bit segments of functions are identified as OVERLAYABLE and are overlaid with other data and bit segments by the BL51 code banking linker/locator.
- **Case/Switch Optimizing**: Depending upon their number, sequence, and location, **switch** and **case** statements can be further optimized by using a jump table or string of jumps.

Options for Code Generation

- **OPTIMIZE(SIZE)**: Common C operations are replaced by subprograms. Program code size is reduced at the expense of program speed.
- **OPTIMIZE(SPEED)**: Common C operations are expanded in-line. Program speed is increased at the expense of code size.
- **NOAREGS**: The C51 compiler no longer uses absolute register access. Program code is independent of the register bank.
- **NOREGPARMS**: Parameter passing is always performed in local data segments rather then dedicated registers. Program code created with this #pragma is compatible to earlier versions of the C51 compiler, the PL/M-51 compiler, and the ASM-51 assembler.

Debugging

The C51 compiler uses the Intel Object Format (OMF51) for object files and generates complete symbol information. Additionally, the compiler can include all the necessary information such as; variable names, function names, line numbers, and so on to allow detailed and thorough debugging and analysis with dScope-51 or Intel compatible emulators. All Intel compatible emulators may be used for program debugging. In addition, the **OBJECTEXTEND** control directive embeds additional variable type information in the object file which allows type-specific display of variables and structures when using certain emulators. You should check with your emulator vendor to determine if it is compatible with the Intel OMF51 object module format and if it can accept Keil object modules.

Library Routines

The C51 compiler includes seven different ANSI compile-time libraries which are optimized for various functional requirements.

Library File	Description
C51S.LIB	Small model library without floating-point arithmetic
C51FPS.LIB	Small model floating-point arithmetic library
C51C.LIB	Compact model library without floating-point arithmetic
C51FPC.LIB	Compact model floating-point arithmetic library
C51L.LIB	Large model library without floating-point arithmetic
C51FPL.LIB	Large model floating-point arithmetic library
80C751.LIB	Library for use with the Philips 8xC751 and derivatives.

Source code is provided for library modules that perform hardware-related I/O and is found in the **\C51\LIB** directory. You may use these source files to help you quickly adapt the library to perform I/O using any I/O device in your target.

Intrinsic Library Routines

The libraries included with the compiler include a number of routines that are implemented as intrinsic functions. Non-intrinsic functions generate **ACALL** or **LCALL** instructions to perform the library routine. Intrinsic functions generate in-line code (which is faster and more efficient) to perform the library routine.

Intrinsic Function	Description
crol	Rotate character left.
cror	Rotate character right.
irol	Rotate integer left.
iror	Rotate integer right.
lrol	Rotate long integer left.
lror	Rotate long integer right.
nop	No operation (8051 NOP instruction).
testbit	Test and clear bit (8051 JBC instruction).

Program Invocation

Typically, the C51 compiler will be called from the µVision2 IDE when you build your project. However, you may invoke the compiler also within a DOS box by typing C51 on the command line. Additionally the name of the C source file to compile is specified on the invocation line as well as any optional control parameters to affect the way the compiler functions.

Example

```
>C51 MODULE.C COMPACT PRINT (E:M.LST) DEBUG SYMBOLS
C51 COMPILER V6.00

C51 COMPILATION COMPLETE.  0 WARNING(S), 0 ERROR(S)
```

Control directives can also be entered via the ***#pragma*** directive, at the beginning of the C source file. For a list of available C51 directives refer to “C51/C251 Compiler” on page 179.

Sample Program

The following example shows some functional capabilities of C51. The C51 compiler produces object files in OMF-51 format, in response to the various C language statements and other directives.

Additionally and optionally, the compiler can emit all the necessary information such as; variable names, function names, line numbers, and so on to allow detailed program debugging and analysis with the µVision2 Debugger or emulators.

The compilation phase also produces a listing file that contains source code, directive information, an assembly listing, and a symbol table. An example for a listing file created by the C51 compiler is shown on the next page.

```
C51 COMPILER V6.00,  SAMPLE            07/01/99  08:00:00  PAGE 1

DOS C51 COMPILER V6.00, COMPILATION OF MODULE SAMPLE
OBJECT MODULE PLACED IN SAMPLE.OBJ
COMPILER INVOKED BY: C:\KEIL\C51\BIN\C51.EXE SAMPLE.C CODE

stmt level source
   1       #include <reg51.h>   /* SFR definitions for 8051 */
   2       #include <stdio.h>   /* standard i/o definitions */
   3       #include <ctype.h>   /* defs for char conversion */
   4
   5       #define EOT  0x1A    /* Control+Z signals EOT */
   6
   7       void main (void)  {
   8   1      unsigned char c;
   9   1
  10   1      /* setup serial port hdw (2400 Baud @12 MHz) */
  11   1      SCON = 0x52;      /* SCON */
  12   1      TMOD = 0x20;      /* TMOD */
  13   1      TCON = 0x69;      /* TCON */
  14   1      TH1  = 0xF3;      /* TH1  */
  15   1
  16   1      while ((c = getchar ()) != EOF)  {
  17   2         putchar (toupper (c));
  18   2         }
  19   1      P0 = 0;   /* clear Output Port to signal ready */
  20   1      }

ASSEMBLY LISTING OF GENERATED OBJECT CODE

             ; FUNCTION main (BEGIN)
                                           ; SOURCE LINE # 7
                                           ; SOURCE LINE # 11
0000 759852        MOV     SCON,#052H
                                           ; SOURCE LINE # 12
0003 758920        MOV     TMOD,#020H
                                           ; SOURCE LINE # 13
0006 758869        MOV     TCON,#069H
                                           ; SOURCE LINE # 14
0009 758DF3        MOV     TH1,#0F3H
000C         ?C0001:
                                           ; SOURCE LINE # 16
000C 120000  E     LCALL   getchar
000F 8F00    R     MOV     c,R7
0011 EF            MOV     A,R7
0012 F4            CPL     A
0013 6008          JZ      ?C0002
                                           ; SOURCE LINE # 17
0015 120000  E     LCALL   _toupper
0018 120000  E     LCALL   _putchar
                                           ; SOURCE LINE # 18
001B 80EF          SJMP    ?C0001
001D         ?C0002:
                                           ; SOURCE LINE # 19
001D E4            CLR     A
001E F580          MOV     P0,A
                                           ; SOURCE LINE # 20
0020 22            RET
             ; FUNCTION main (END)


MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     33    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----       1
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.


C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

C51 produces a listing file with line numbers as well as the time and date of the compilation.

Information about compiler invocation and the object file generated is printed.

The listing contains a line number before each source line and the instruction nesting { } level.

If errors or possible sources of errors exist an error or warning message is displayed.

Enable under µVision2 ***Options for Target – Listing - Assembly Code*** *the C51* ***CODE*** *directive. This gives you an assembly listing file with embedded source line numbers.*

A51 Macro Assembler

A51 is a macro assembler for the 8051 microcontroller family. A51 translates symbolic assembler language mnemonics into executable machine code. A51 allows you to define each instruction in a 8051 program and is used where utmost speed, small code size and exact hardware control is essential. The A51 macro facility saves development and maintenance time, since common sequences need only be developed once.

Source-Level Debugging

A51 generates complete symbol and type information; this allows an exact display of program variables. Even line numbers of the source file are available to enable source level debugging for assembler programs with the µVision2 Debugger or emulators.

Functional Overview

A51 translates an assembler source file into a relocatable object module. A51 generates a listing file, optionally with symbol table and cross reference. A51 contains two macro processors:

- **Standard Macros** are simple to use and enable you to define and to use macros in your 8051 assembly programs. The standard macros are used in many assemblers.
- The **Macro Processing Language** (MPL) is a string replacement facility. It is fully compatible with Intel ASM51 and has several predefined macro processor functions. These MPL processor functions perform many useful operations, like string manipulation or number processing.

Another powerful feature of A51 macro assembler is conditional assembly depending on command line directives or assembler symbols. Conditional assembly of sections of code can help you to achieve the most compact code possible or to generate different applications out of one assembly source file.

Listing File

On the following page is an example listing file generated by the assembler.

```
A51 MACRO ASSEMBLER  Test Program        07/01/99 08:00:00 PAGE     1

DOS MACRO ASSEMBLER A51 V6.00
OBJECT MODULE PLACED IN SAMPLE.OBJ
ASSEMBLER INVOKED BY: C:\KEIL\C51\BIN\A51.EXE SAMPLE.A51 XREF

LOC  OBJ       LINE  SOURCE
                  1  $TITLE ('Test Program')
                  2  NAME    SAMPLE
                  3
                  4  EXTRN CODE (PUT_CRLF, PUTSTRING, InitSerial)
                  5  PUBLIC  TXTBIT
                  6
                  7  PROG    SEGMENT         CODE
                  8  CONST   SEGMENT         CODE
                  9  BITVAR  SEGMENT         BIT
                 10
----             11          CSEG  AT     0
                 12
0000 020000   F  13  Reset:  JMP   Start
                 14
----             15          RSEG  PROG
                 16   ; *****
0000 120000   F  17  Start:  CALL  InitSerial ;Init Serial Interface
                 18
                 19  ; This is the main program. It is an endless
                 20  ; loop which displays a text on the console.
0003 C200     F  21          CLR   TXTBIT       ; read from CODE
0005 900000   F  22  Repeat: MOV   DPTR,#TXT
0008 120000   F  23          CALL  PUTSTRING
000B 120000   F  24          CALL  PUT_CRLF
000E 80F5        25          SJMP  Repeat
                 26  ;
----             27          RSEG  CONST
0000 54455354    28  TXT:    DB    'TEST PROGRAM',00H
0004 2050524F
0008 4752414D
000C 00
                 29
                 30
                 31
----             32          RSEG  BITVAR  ; TXTBIT=0 read from CODE
0000             33  TXTBIT: DBIT  1       ; TXTBIT=1 read from XDATA
                 34
                 35          END

XREF SYMBOL TABLE LISTING
---- ------ ----- -------

N A M E             T Y P E  V A L U E   ATTRIBUTES / REFERENCES

BITVAR . . . . . .  B SEG    0001H       REL=UNIT   9# 32
CONST. . . . . . .  C SEG    000DH       REL=UNIT   8# 27
INITSERIAL . . . .  C ADDR   -----       EXT   4# 17
PROG . . . . . . .  C SEG    0010H       REL=UNIT   7# 15
PUTSTRING. . . . .  C ADDR   -----       EXT   4# 23
PUT_CRLF . . . . .  C ADDR   -----       EXT   4# 24
REPEAT . . . . . .  C ADDR   0005H   R   SEG=PROG    22# 25
RESET. . . . . . .  C ADDR   0000H   A      13#
SAMPLE . . . . . .  N NUMB   -----          2
START. . . . . . .  C ADDR   0000H   R   SEG=PROG   13 17#
TXT. . . . . . . .  C ADDR   0000H   R   SEG=CONST   22 28#
TXTBIT . . . . . .  B ADDR   0000H.0 R   SEG=BITVAR   5 5 21 33#

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE.  0 WARNING(S), 0 ERROR(S)
```

A51 produces a listing file with line numbers as well as the time and date of the translation. Information about assembler invocation and the object file generated is printed.

BL51 Code Banking Linker/Locator

The BL51 code banking linker/locator combines one or more object modules into a single executable 8051 program. The linker also resolves external and public references, and assigns absolute addresses to relocatable programs segments.

The BL51 code banking linker/locator processes object modules created by the Keil C51 compiler and A51 assembler and the Intel PL/M-51 compiler and ASM-51 assembler. The linker automatically selects the appropriate run-time library and links only the library modules that are required.

Normally, you invoke the BL51 code banking linker/locator from the command line specifying the names of the object modules to combine. The default controls for the BL51 code banking linker/locator have been carefully chosen to accommodate most applications without the need to specify additional directives. However, it is easy for you to specify custom settings for your application.

Data Address Management

The BL51 code banking linker/locator manages the limited internal memory of the 8051 by overlaying variables for functions that are mutually exclusive. This greatly reduces the overall memory requirement of most 8051 applications.

The BL51 code banking linker/locator analyzes the references between functions to carry out memory overlaying. You may use the **OVERLAY** directive to manually control functions references the linker uses to determine exclusive memory areas. The **NOOVERLAY** directive lets you completely disable memory overlaying. These directives are useful when using indirectly called functions or when disabling overlaying for debugging.

Code Banking

The BL51 code banking linker/locator supports the ability to create application programs that are larger than 64 Kbytes. Since the 8051 does not directly support more than 64 Kbytes of code address space, there must be external hardware that swaps code banks. The hardware that does this must be controlled by software running on the 8051. This process is known as bank switching.

The BL51 code banking linker/locator lets you manage 1 common area and 32 banks of up to 64 Kbytes each for a total of 2 Mbytes of bank-switched 8051 program space. Software support for the external bank switching hardware includes a short assembly file you can edit for your specific hardware platform.

The BL51 code banking linker/locator lets you specify the bank in which to locate a particular program module. By carefully grouping functions in the different banks, you can create very large, efficient applications.

Common Area

The common area in a bank switching program is an area of memory that can be accessed at all times from all banks. The common area cannot be physically swapped out or moved around. The code in the common area is either duplicated in each bank (if the entire program area is swapped) or can be located in a separate area or EPROM (if the common area is not swapped).

The common area contains program sections and constants which must be available at all times. It may also contain frequently used code. By default, the following code sections are automatically located in the common area:

- Reset and Interrupt Vectors,
- Code Constants,
- C51 Interrupt Functions,
- Bank Switch Jump Table,
- Some C51 Run-Time Library Functions.

Executing Functions in Other Banks

Code banks are selected by additional software-controlled address lines that are simulated using 8051 port I/O lines or a memory-mapped latch.

The BL51 code banking linker/locator generates a jump table for functions in other code banks. When your C program calls a function located in a different bank, it switches the bank, jumps to the desired function, restores the previous bank (when the function completes), and returns execution to the calling routine.

The bank switching process requires approximately 50 CPU cycles and consumes an additional 2 bytes of stack space. You can dramatically improve system performance by grouping interdependent functions in the same bank. Functions which are frequently invoked from multiple banks should be located in the common area.

Map File

On the following page is an example listing file generated by BL51.

```
BL51 BANKED LINKER/LOCATER V4.00        07/01/99  08:00:00  PAGE 1

MS-DOS BL51 BANKED LINKER/LOCATER V4.00, INVOKED BY:
C:\KEIL\C51\BIN\BL51.EXE SAMPLE.OBJ

MEMORY MODEL: SMALL

INPUT MODULES INCLUDED:
  SAMPLE.OBJ (SAMPLE)
  C:\C51\LIB\C51S.LIB (?C_STARTUP)
  C:\C51\LIB\C51S.LIB (PUTCHAR)
  C:\C51\LIB\C51S.LIB (GETCHAR)
  C:\C51\LIB\C51S.LIB (TOUPPER)
  C:\C51\LIB\C51S.LIB (_GETKEY)


LINK MAP OF MODULE:  SAMPLE (SAMPLE)

     TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
     -----------------------------------------------------

     * * * * * * *   D A T A   M E M O R Y   * * * * * * *
     REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
     DATA    0008H     0001H     UNIT         ?DT?GETCHAR
     DATA    0009H     0001H     UNIT         _DATA_GROUP_
             000AH     0016H                  *** GAP ***
     BIT     0020H.0   0000H.1   UNIT         ?BI?GETCHAR
             0020H.1   0000H.7                *** GAP ***
     IDATA   0021H     0001H     UNIT         ?STACK

     * * * * * * *   C O D E   M E M O R Y   * * * * * * *
     CODE    0000H     0003H     ABSOLUTE
     CODE    0003H     0021H     UNIT         ?PR?MAIN?SAMPLE
     CODE    0024H     000CH     UNIT         ?C_C51STARTUP
     CODE    0030H     0027H     UNIT         ?PR?PUTCHAR?PUTCHAR
     CODE    0057H     0011H     UNIT         ?PR?GETCHAR?GETCHAR
     CODE    0068H     0018H     UNIT         ?PR?_TOUPPER?TOUPPER
     CODE    0080H     000AH     UNIT         ?PR?_GETKEY?_GETKEY


OVERLAY MAP OF MODULE:   SAMPLE (SAMPLE)

SEGMENT                           DATA_GROUP
  +--> CALLED SEGMENT           START    LENGTH
----------------------------------------------
?C_C51STARTUP                   -----    -----
  +--> ?PR?MAIN?SAMPLE

?PR?MAIN?SAMPLE                 0009H    0001H
  +--> ?PR?GETCHAR?GETCHAR
  +--> ?PR?_TOUPPER?TOUPPER
  +--> ?PR?PUTCHAR?PUTCHAR

?PR?GETCHAR?GETCHAR             -----    -----
  +--> ?PR?_GETKEY?_GETKEY
  +--> ?PR?PUTCHAR?PUTCHAR


LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

BL51 produces a MAP file (extension .M51) with date and time of the link/locate run.

BL51 displays the invocation line and the memory model.

Each input module and the library modules included in the application are listed.

The memory map contains the usage of the physical 8051 memory.

The overlay-map displays the structure of the program and the location of the bit and data segments of each function.

Warning messages and error messages are listed at the end of the MAP file. These may point to possible problems encountered during the link/locate run.

LIB51 Library Manager

The LIB51 library manager lets you create and maintain library files. A library file is a formatted collection of object modules (created by the C compiler and assembler). Library files provide a convenient method of combining and referencing a large number of object modules that may be accessed by the linker/locator.

To build a library with the µVision2 project manager enable **Options for Target – Output – Create Library**. You may also call **LIB51** from a DOS box. Refer to "LIB51 / L251 Library Manager Commands" on page 185 for command list.

There are a number of benefits to using a library. Security, speed, and minimized disk space are only a few of the reasons to use a library. Additionally, libraries provide a good vehicle for distributing a large number of useful functions and routines without the need to distribute source code. For example, the ANSI C library is provided as a set of library files.

The µVision2 project **C:\KEIL\C51\RTX_TINY\RTX_TINY.UV2** allows you to modify and create the RTX51 Tiny real-time operating system library. It is easy to build your own library of useful routines like serial I/O, CAN, and FLASH memory utilities that you may use over and over again. Once these routines are written and debugged, you may merge them into a library. Since the library contains only the object modules, the build time is shortened since these modules do not require re-compilation for each project.

Libraries are used by the linker when linking and locating the final application. Modules in the library are extracted and added to the program only if they are required. Library routines that are not specifically invoked by your program are not included in the final output. The linker extracts the modules from the library and processes them exactly as it does other object modules.

OC51 Banked Object File Converter

The OC51 banked object file converter creates absolute object modules for each code bank in a banked object module. Banked object modules are created by the BL51 code banking linker/locator when you create a bank switching application. Symbolic debugging information is copied to the absolute object files and can be used by dScope or an in-circuit emulator.

You may use the OC51 banked object file converter to create absolute object modules for the command area and for each code bank in your banked object module. You may then generate Intel HEX files for each of the absolute object modules using the OH51 object-hex converter.

OH51 Object-Hex Converter

The OH51 object-hex converter creates Intel HEX files from absolute object modules. Absolute object modules can be created by the BL51 code banking linker or by the OC51 banked object file converter. Intel HEX files are ASCII files that contain a hexadecimal representation of your application. They can be easily loaded into a device programmer for writing EPROMS.

Chapter 4. Creating Applications

To make it easy for you to evaluate and become familiar with our 166 product line, we provide an evaluation version with sample programs and limited versions of our tools. The sample programs are also included with our standard product kits.

NOTE
The Keil C51 evaluation tools are limited in functionality and the code size of the application you can create. Refer to the "Release Notes" for more information on the limitations of the evaluation tools. For larger applications, you need to purchase one of our development kits. Refer to "Product Overview" on page 8 for a description of the kits that are available.

This chapter describes the **Build Mode** of µVision2 and shows you how to use the user interface to create a sample program. Also discussed are options for generating and maintaining projects. This includes output file options, the configuration of the C51 compiler for optimum code quality, and the features of the µVision2 project manager.

Create a Project

µVision2 includes a project manager which makes it easy to design applications for the 8051 family. You need to perform the following steps to create a new project:

- Start µVision2, create a project file and select a CPU from the device database.
- Create a new source file and add this source file to the project.
- Add and configure the startup code for the 8051 device
- Set tool options for target hardware.
- Build project and create a HEX file for PROM programming.

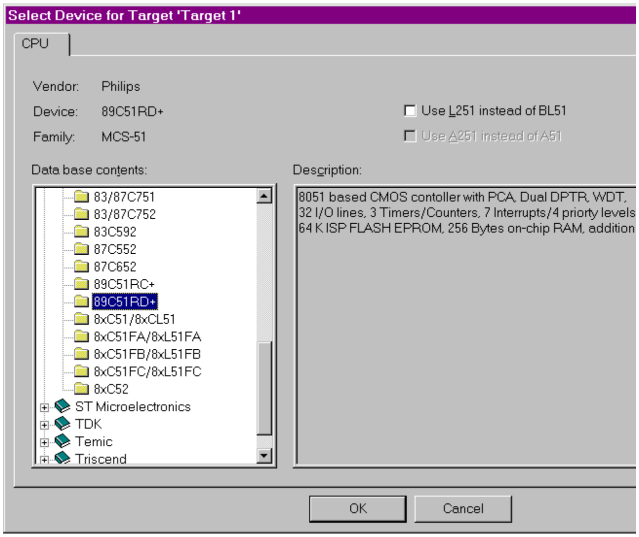
The description is a step-by-step tutorial that shows you how to create a simple µVision2 project.

Start µVision2 and Create a Project File

µVision2 is a standard Windows application and started by clicking on the program icon. To create a new project file select from the µVision2 menu **Project – New Project…**. This opens a standard Windows dialog that asks you for the new project file name.

We suggest that you use a separate folder for each project. You can simply use the icon **Create New Folder** in this dialog to get a new empty folder. Then select this folder and enter the file name for the new project, i.e. **Project1**. µVision2 creates a new project file with the name **PROJECT1.UV2** which contains a default target and file group name. You can see these names in the **Project Window – Files**.

Now use from the menu **Project – Select Device for Target** and select a CPU for your project. The **Select Device** dialog box shows the µVision2 device database. Just select the microcontroller you use. We are using for our examples the Philips 80C51RD+ CPU. This selection sets necessary tool options for the 80C51RD+ device and simplifies in this way the tool configuration.

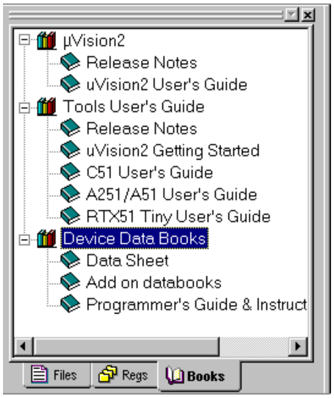

Select Device for Target 'Target 1'

CPU

Vendor: Philips
Device: 89C51RD+
Family: MCS-51

☐ Use L251 instead of BL51
☐ Use A251 instead of A51

Description:
8051 based CMOS contoller with PCA, Dual DPTR, WDT,
32 I/O lines, 3 Timers/Counters, 7 Interrupts/4 priorty levels
64 K ISP FLASH EPROM, 256 Bytes on-chip RAM, addition

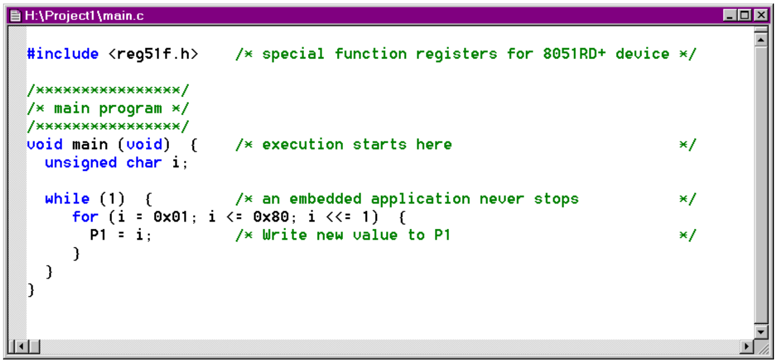


NOTE
On some devices, the µVision2 environment needs additional parameters that you have to enter manually. Please carefully read the information provided under ***Description*** *in this dialog, since it might have additional instructions for the device configuration.*

Once you have selected a CPU from the device database you can open the user manuals for that device in the **Project Window – Books** page. These user manuals are part of the Keil Development Tools CD-ROM that should be present in your CD drive.

Create New Source Files

You may create a new source file with the menu option **File – New**. This opens an empty editor window where you can enter your source code. µVision2 enables the C color syntax highlighting when you save your file with the dialog **File – Save As…** under a filename with the extension ***.C**. We are saving our example file under the name **MAIN.C**.



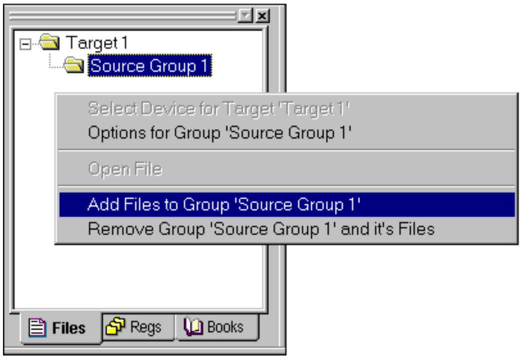
H:\Project1\main.c

```
#include <reg51f.h>    /* special function registers for 8051RD+ device */

/****************/
/* main program */
/****************/
void main (void)  {    /* execution starts here                         */
  unsigned char i;

  while (1)  {         /* an embedded application never stops           */
     for (i = 0x01; i <= 0x80; i <<= 1)  {
       P1 = i;         /* Write new value to P1                         */
     }
  }
}
```

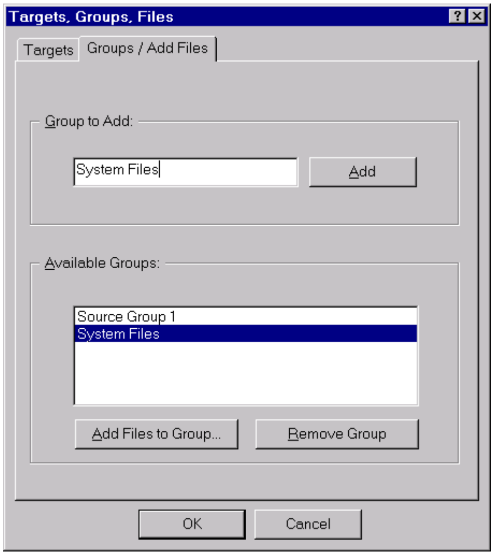
Once you have created your source file you can add this file to your project. µVision2 offers several ways to add source files to a project. For example, you can select the file group in the **Project Window – Files** page and click with the right mouse key to open a local menu. The option **Add Files** opens the standard files dialog. Select the file **MAIN.C** you have just created.


Target 1
Source Group 1
Select Device for Target 'Target 1'
Options for Group 'Source Group 1'
Open File
Add Files to Group 'Source Group 1'
Remove Group 'Source Group 1' and it's Files
Files
Regs
Books

Add and Configure the Startup Code

The **STARTUP.A51** file is the startup code for the most 8051 CPU variants. The startup code clears the data memory and initializes hardware and reentrant stack pointers. In addition, some 8051 derivatives require a CPU initialization code that needs to match the configuration of your hardware design. For example, the Philips 8051RD+ offers you on-chip xdata RAM that should be enabled in the startup code. Since you need to modify that file to match your target hardware, you should copy the **STARTUP.A51** file from the folder **C:\KEIL\C51\LIB** to your project folder.

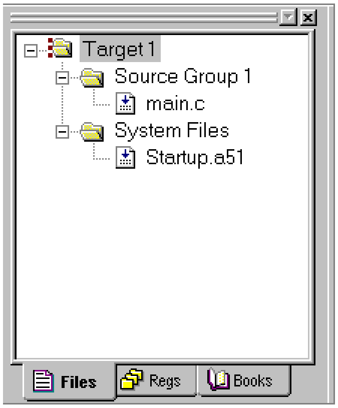
It is a good practice to create a new file group for the CPU configuration files. With **Project – Targets, Groups, Files…** you can open a dialog box where you add a group named **System Files** to your target. In the same dialog box you can use the **Add Files to Group…** button to add the **STARTUP.A51** file to your project.



Group to Add: System Files

Available Groups:
Source Group 1
System Files

The **Project Window – Files** lists all items of your project.

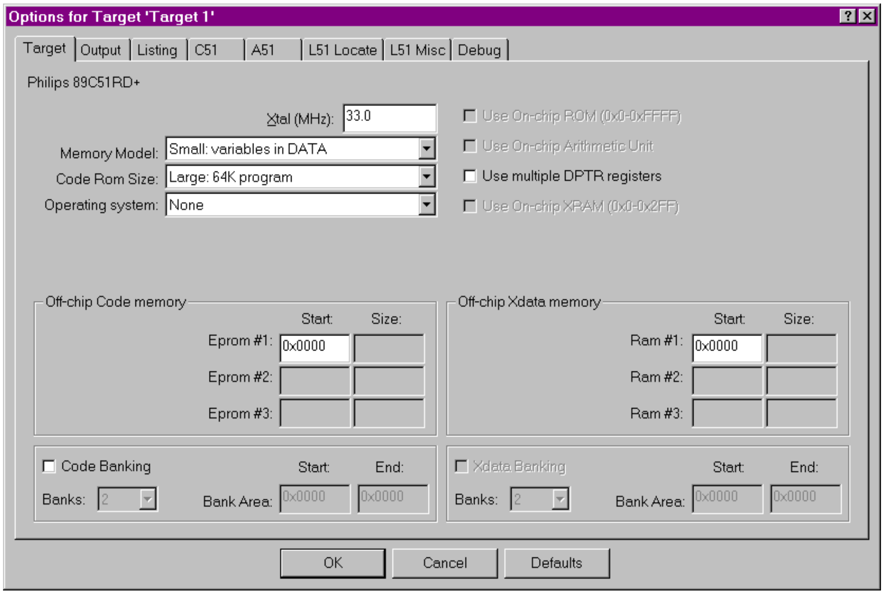


Target 1
Source Group 1
main.c
System Files
Startup.a51

The µVision2 **Project Window – Files** should now show the above file structure. Open **STARTUP.A51** in the editor with a double click on the file name in the project window. Then you configure the startup code as described in "Chapter 10. CPU and C Startup Code" on page 173. If you are using on-chip RAM of your device the settings in the startup code should match the settings of the **Options – Target** dialog. This dialog is discussed in the following.

Set Tool Options for Target

µVision2 lets you set options for your target hardware. The dialog **Options for Target** opens via the toolbar icon. In the **Target** tab you specify all relevant parameters of your target hardware and the on-chip components of the device you have selected. The following the settings for our example are shown.



Options for Target 'Target 1'

Target | Output | Listing | C51 | A51 | L51 Locate | L51 Misc | Debug

Philips 89C51RD+

Xtal (MHz): 33.0

Memory Model: Small: variables in DATA

Code Rom Size: Large: 64K program

Operating system: None

□ Use On-chip ROM (0x0-0xFFFF)

□ Use On-chip Arithmetic Unit

□ Use multiple DPTR registers

□ Use On-chip XRAM (0x0-0x2FF)

Off-chip Code memory

	Start:	Size:
Eprom #1:	0x0000	
Eprom #2:		
Eprom #3:		

Off-chip Xdata memory

	Start:	Size:
Ram #1:	0x0000	
Ram #2:		
Ram #3:		

□ Code Banking — Banks: 2 — Bank Area: Start: 0x0000 End: 0x0000

□ Xdata Banking — Banks: 2 — Bank Area: Start: 0x0000 End: 0x0000

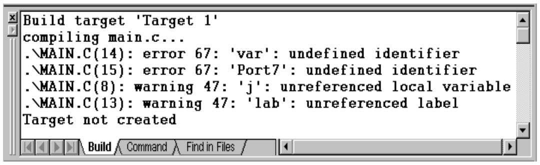
The following table describes the options of the **Target** dialog:

Dialog Item	Description
Xtal	specifies the CPU clock of your device. In most cases this value is identical with the XTAL frequency.
Memory Model	specifies the C51 compiler memory model. For starting new applications the default **SMALL** is a good choice. Refer to "Memory Models and Memory Types" on page 67 for a discussion of the various memory models.
Allocate On-chip … Use multiple DPTR registers	specifies the usage of the on-chip components which are typically enabled in the CPU startup code. If you are using on-chip xdata RAM (XRAM) you should also enable the XRAM access in the STARTUP.A51 file.
Off-chip … Memory	here you specify all external memory areas of the target hardware.
Code Banking Xdata Banking	specifies the parameters for code and xdata banking.

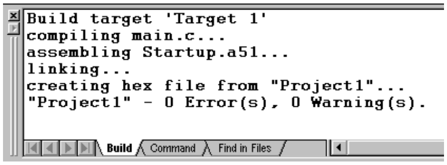
NOTE
Several Options in the Target dialog are only available if you are using the L251 Linker/Locater. The L251 Linker/Locater is available for the 8051 within the C51 Variable Banking extension that is available as add-on product to the C51 Compiler. Check www.keil.com for details.

Build Project and Create a HEX File

Typical, the tool settings under **Options – Target** are all you need to start a new application. You may translate all source files and line the application with a click on the **Build Target** toolbar icon. When you build an application with syntax errors, µVision2 will display errors and warning messages in the **Output Window – Build** page. A double click on a message line opens the source file on the correct location in a µVision2 editor window.

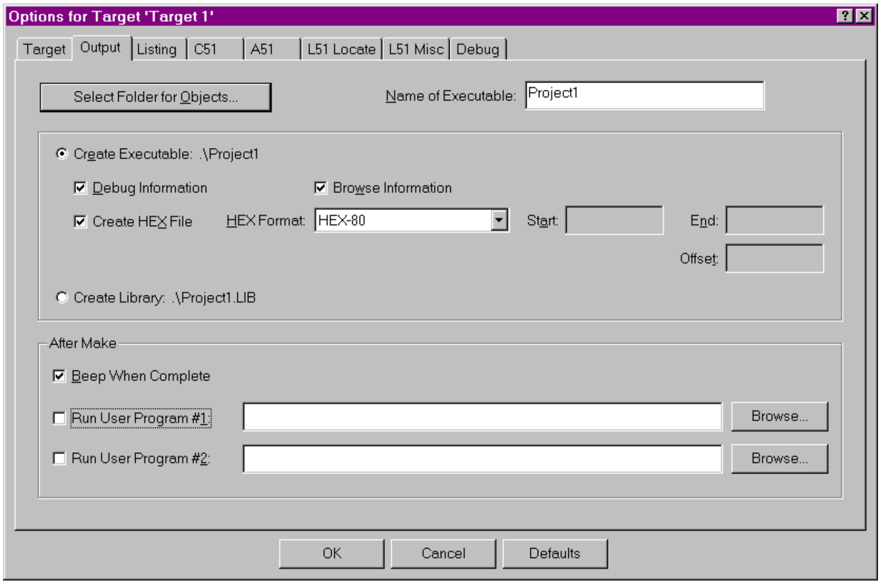


```
Build target 'Target 1'
compiling main.c...
.\MAIN.C(14): error 67: 'var': undefined identifier
.\MAIN.C(15): error 67: 'Port7': undefined identifier
.\MAIN.C(8): warning 47: 'j': unreferenced local variable
.\MAIN.C(13): warning 47: 'lab': unreferenced label
Target not created
```



```
Build target 'Target 1'
compiling main.c...
assembling Startup.a51...
linking...
creating hex file from "Project1"...
"Project1" - 0 Error(s), 0 Warning(s).
```

Once you have successfully generated your application you can start debugging. Refer to "Chapter 5. Testing Programs" on page 81 for a discussion of the µVision2 debugging features. After you have tested your application, it is required to create an Intel HEX file to download the software into an EPROM programmer or simulator. µVision2 creates HEX files with each build process when **Create HEX file** under **Options for Target – Output** is enabled. The **Start**, **End** and **Offset** values are only available if you are using the L251 Linker/Locater. You may start your PROM programming utility after the make process when you specify the program under the option **Run User Program #1.**



Options for Target 'Target 1'

Name of Executable: Project1

● Create Executable: .\Project1

☑ Debug Information ☑ Browse Information

☑ Create HEX File HEX Format: HEX-80 Start: End: Offset:

○ Create Library: .\Project1.LIB

After Make

☑ Beep When Complete

☐ Run User Program #1:

☐ Run User Program #2:

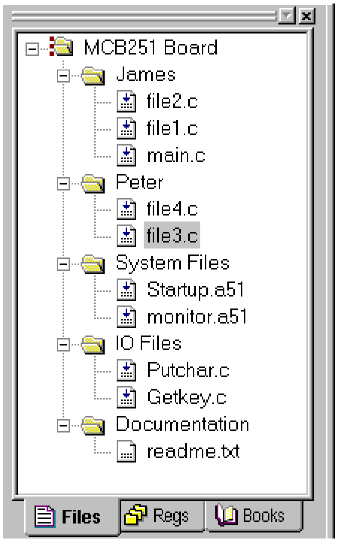
Now you can modify existing source code or add new source files to the project. The **Build Target** toolbar button translates only modified or new source files and generates the executable file. µVision2 maintains a file dependency list and knows all include files used within a source file. Even the tool options are saved in the file dependency list, so that µVision2 rebuilds files only when needed. With the **Rebuild Target** command, all source files are translated, regardless of modifications.

Project Targets and File Groups

By using different **Project Targets** µVision2 lets you create several programs from a single project. You may need one target for testing and another target for a release version of your application. Each target allows individual tool settings within the same project file.

Files Groups let you group associated files together in a project. This is useful for grouping files into functional blocks or for identifying engineers in your software team. We have already used file groups in our example to separate the CPU related files from other source files. With these technique it is easily possible to maintain complex projects with several 100 files in µVision2.

The **Project – Targets, Groups, Files…** dialog allows you to create project targets and file groups. We have already used this dialog to add the system configuration files. An example project structure is shown below.

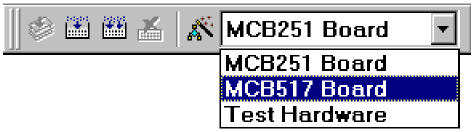


- MCB251 Board
  - James
    - file2.c
    - file1.c
    - main.c
  - Peter
    - file4.c
    - file3.c
  - System Files
    - Startup.a51
    - monitor.a51
  - IO Files
    - Putchar.c
    - Getkey.c
  - Documentation
    - readme.txt

The **Project Windows** shows all groups and the related files. Files are built and linked in the same order as shown in this window. You can move file positions with **Drag & Drop**. You may select a target or group name and **Click** to rename it. The local menu opens with a right mouse **Click** and allows you for each item:

- to set tool options
- to remove the item
- to add files to a group
- to open the file.

In the build toolbar you can quickly change the current project target to build.



View File and Group Attributes in the Project Window

Different icons are used in the **Project Window – Files** page to show the attributes of files and file groups. These icons are explained below:

Files that are translated and linked into the project are marked with an arrow in the file icon.

Files that are excluded from the link run do not have the arrow. This is typical for document files. However you may exclude also source files when you disable **Include in Target Build** in the **Properties** dialog. See also “File and Group Specific Options – Properties Dialog” on page 75.

Read only files are marked with a key. This is typical for files that are checked into a Software Version Control System, since the SVCS makes the local copy of such files read only. See also “Using the SVCS Menu” on page 64.

Files or file groups with specific options are marked with dots. Refer to “File and Group Specific Options – Properties Dialog” on page 75 for more information.

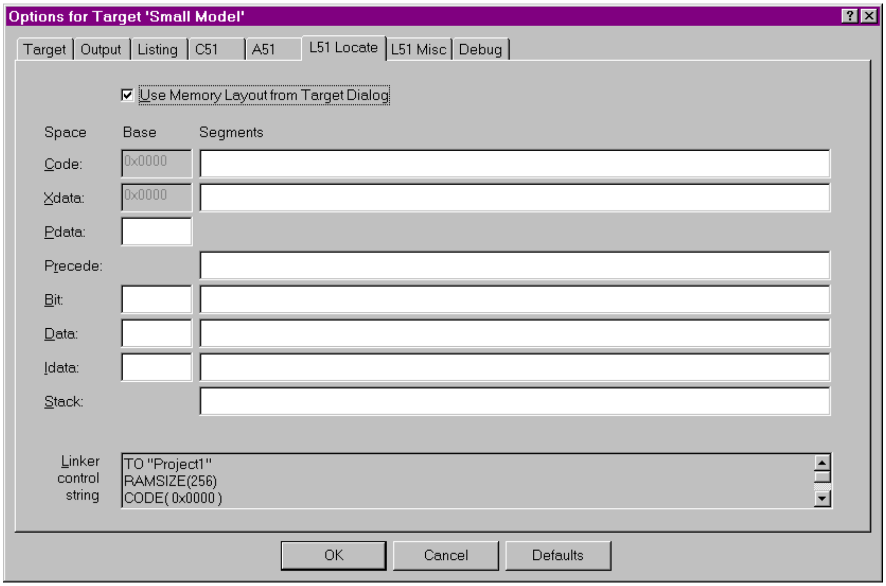
NOTE
The different icons give you quick overview of the tool settings in the various targets of a project. The icons reflect always the attributes of the current selected target. For example, if you have set specific options on a file or file group in one target, then the dots in the icon are only shown if this target is currently selected.

Overview of Configuration Dialogs

The options dialog lets you set all the tool options. Via the local menu in the **Project Window – Files** you may set different options for a file group or even a single file; in this case you get only the related dialog pages. With the context help button ? you get help on most dialog items. The following table describes the options of the **Target** dialog.

Dialog Page	Description
Target	Specify the hardware of your application. See page 52 for details.
Output	Define the output files of the tool chain and allows you to start user programs after the build process. See page 70 for more information.
Listing	Specify all listing files generated by the tool chain.
C51	Set C51 compiler specific tool options like code optimization or variable allocation. Refer to “Other C51 Compiler Directives” on page 68 for information.
A51 A251	Set assembler specific tool options like macro processing.
L51 Locate L251 Locate	Define the location of memory classes and segments. Typical you will enable **Use Memory Layout from Target Dialog** as show below to get automatic settings. Refer to “Locate Segments to Absolute Memory Locations” on page 74 for more information on this dialog.
L51 Misc L251 Misc	Other linker related settings like **Warning** or memory **Reserve** directive. You need to reserve some memory locations when you are using Monitor-51 for debugging.
Debug	Settings for the µVision2 Debugger. Refer to page 88 for more information.
Properties	File information and special options for files and groups refer to “File and Group Specific Options – Properties Dialog” on page 75.

Below the **L51 Locate** dialog page is shown. When you enable **Use Memory Layout from Target Dialog** µVision2 uses the memory information from the selected Device and the Target page. You may still add additional segments to these settings.



Options for Target 'Small Model'

☑ Use Memory Layout from Target Dialog

Space	Base	Segments
Code:	0x0000	
Xdata:	0x0000	
Pdata:		
Precede:		
Bit:		
Data:		
Idata:		
Stack:		

Linker control string:

```
TO "Project1"
RAMSIZE(256)
CODE( 0x0000 )
```

µVision2 Utilities

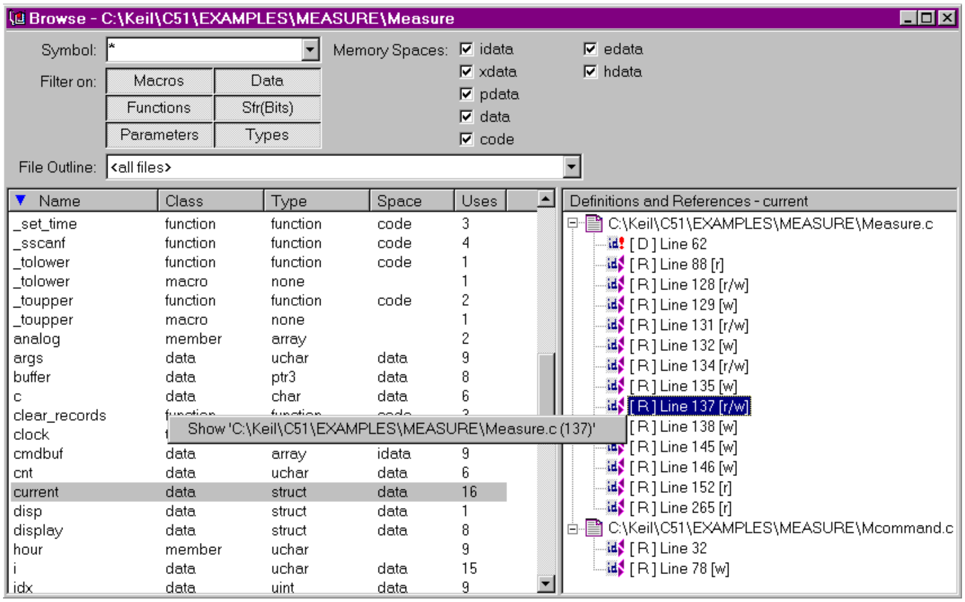
µVision2 contains many powerful functions that help you during your software project. These utilities are discussed in the following section.

Find in Files

The **Edit – Find in Files** dialog performs a global text search in all specified files. The search results are displayed in the Find in Files page of the Output window. A double click in the Find in Files page positions the editor to the text line with matching string.

Source Browser

The Source Browser displays information about program symbols in your program. If **Options for Target – Output – Browser Information** is enabled when you build the target program, the compiler includes browser information into the object files. Use **View – Source Browser** to open the Browse window.


Browse - C:\Keil\C51\EXAMPLES\MEASURE\Measure
Filter on:
Macros
Data
Functions
Sfr(Bits)
Parameters
Types
Definitions and References - current

Symbol: *

Memory Spaces: ☑ idata ☑ xdata ☑ pdata ☑ data ☑ code ☑ edata ☑ hdata

File Outline: <all files>

Name	Class	Type	Space	Uses
_set_time	function	function	code	3
_sscanf	function	function	code	4
_tolower	function	function	code	1
_tolower	macro	none		1
_toupper	function	function	code	2
_toupper	macro	none		1
analog	member	array		2
args	data	uchar	data	9
buffer	data	ptr3	data	8
c	data	char	data	6
clear_records	function	function	code	3
clock	f			
cmdbuf	data	array	idata	9
cnt	data	uchar	data	6
current	data	struct	data	16
disp	data	struct	data	1
display	data	struct	data	8
hour	member	uchar		9
i	data	uchar	data	15
idx	data	uint	data	9

Show 'C:\Keil\C51\EXAMPLES\MEASURE\Measure.c (137)'

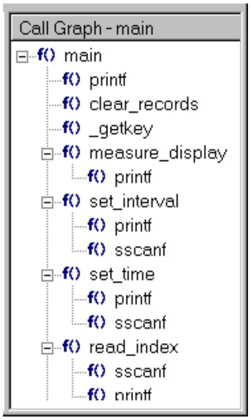
- C:\Keil\C51\EXAMPLES\MEASURE\Measure.c
  - [D] Line 62
  - [R] Line 88 [r]
  - [R] Line 128 [r/w]
  - [R] Line 129 [w]
  - [R] Line 131 [r/w]
  - [R] Line 132 [w]
  - [R] Line 134 [r/w]
  - [R] Line 135 [w]
  - [R] Line 137 [r/w]
  - [R] Line 138 [w]
  - [R] Line 145 [w]
  - [R] Line 146 [w]
  - [R] Line 152 [r]
  - [R] Line 265 [r]
- C:\Keil\C51\EXAMPLES\MEASURE\Mcommand.c
  - [R] Line 32
  - [R] Line 78 [w]

The Browse window lists the symbol name, class, type, memory space and the number of uses. Click on the list item to sort the information. You can filter the browse information using the options described in the following table:

Browse Options	Description
Symbol	specify a mask that is used to match symbol names. The mask may consist of alphanumeric characters plus mask characters: **#** matches a digit (0 – 9) **$** matches any character ***** matches zero or more characters.
Filter on	select the definition type of the symbol
File Outline	select the file where information should be listed for.
Memory Spaces	specify the memory type for data and function symbols.

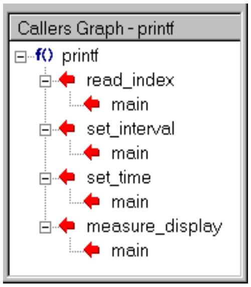
The following table provides a few examples of symbol name masks.

Mask	Matches symbol names …
*****	Matches any symbol. This is the default mask in the Symbol Browser.
#	… that contain one digit in any position.
_a$#*	… with an underline, followed by the letter **a**, followed by any character, followed by a digit, ending with zero or more characters. For example, **_ab1** or **_a10value**.
_*ABC	… with an underline, followed by zero or more characters, followed by **ABC**.



The local menu in the Browse window opens with a right mouse Click and allows you to open the editor on the selected reference. For functions you can also view the Call and Callers graph. The Definitions and References view gives you additional information with the following symbols:

Symbol	Description
[D]	Definition
[R]	Reference
[r]	read access
[w]	write access
[r/w]	read/write access
[&]	address reference



You may use the browser information within an editor window. Select the item that you want to search for and open the local menu with a right mouse click or use the following keyboard shortcuts:

Shortcut	Description
F12	Goto Definition; place cursor to the symbol definition
Shift+F12	Goto Reference; place cursor to a symbol reference
Ctrl+Num+	Goto Next Reference or Definition
Ctrl+Num–	Goto Previous Reference or Definition

Key Sequence for Tool Parameters

A key sequence may be used to pass arguments from the µVision2 environment to external user programs. Key sequences can be applied in the **Tools** menu, **SVCS** menu, and the **Run User Program** arguments in the **Options for Target – Output** dialog. A key sequence is a combination of a **Key Code** and a **File Code**. The available **Key Codes** and **File Codes** are listed in the tables below:

Key Code	Specifies the path selected with the File Code
%	filename with extension, but without path specification (PROJECT1.UV2)
#	filename with complete path specification (C:\MYPROJECT\PROJECT1.UV2)
%	filename with extension, but without path specification (PROJECT1.UV2)
@	filename without extension and path specification (PROJECT1)
$	folder name of the file specified in the file code (C:\MYPROJECT)
~ †	line number of current cursor position (only valid for file code **F**)
^ †	column number of current cursor position (only valid for file code **F**)

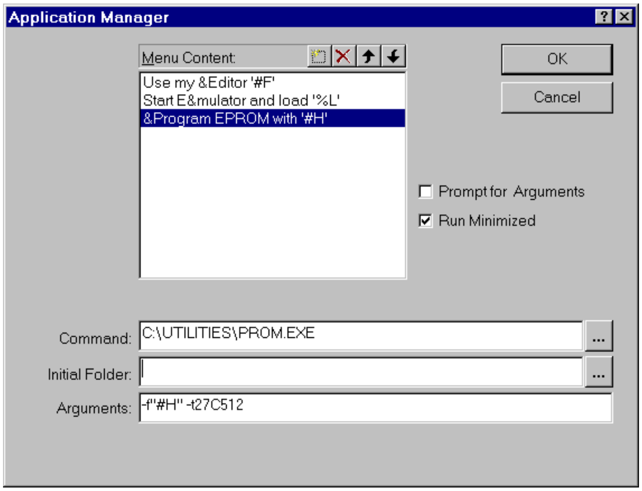
† the key code ~ and ^ can be used only in combination with the file code **F**
To use $, #, %, @, ~ or ^ in the user program command line, use $$, ##, %%, @@, ~~ or ^^.
For example @@ gives a single @ in the user program command line.

File Code	Specifies the file name or argument inserted in the user program line
F	selected file in the **Project Window - Files** page (MEASURE.C). Returns the project file if the target name is selected or the current active editor file if a group name is selected.
P	name of the current project file (PROJECT1.UV2)
L	linker output file, typical the executable file for debugging (PROJECT1)
H	application HEX file (PROJECT1.H86)
X	µVision2 executable program file (C:\KEIL\UV2\UV2.EXE)
	The following file codes are used for SVCS systems. For more information refer to "Using the SVCS Menu" on page 64.
Q †	file name that holds comments for the SVCS system.
R †	string that holds a revision number for the SVCS system.
C †	string that holds a check point string for the SVCS system.
U †	user name specified under **SVCS – Configure Version Control – User Name**
V †	file name specified under **SVCS – Configure Version Control – Database**

† the file codes Q, R, C, U and V can be used only in combination with the key code %

Using the Tools Menu

Via the **Tools** menu, you run external programs. You may add custom programs to the **Tools** menu with the dialog **Tools – Customize Tools Menu…**. This dialog configures the parameters for external user applications. The dialog right shows a sample tool setup. The dialog options are explained in the table below.


Application Manager

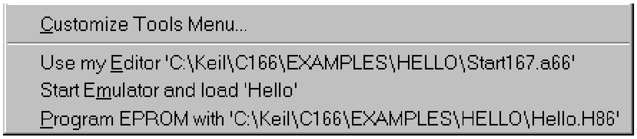
Menu Content:

Use my &Editor '#F'
Start E&mulator and load '%L'
&Program EPROM with '#H'

☐ Prompt for Arguments
☑ Run Minimized

Command: C:\UTILITIES\PROM.EXE
Initial Folder:
Arguments: -f"#H" -t27C512

The above entries extend the **Tools** menu as shown right.


Customize Tools Menu...
Use my Editor 'C:\Keil\C166\EXAMPLES\HELLO\Start167.a66'
Start Emulator and load 'Hello'
Program EPROM with 'C:\Keil\C166\EXAMPLES\HELLO\Hello.H86'

Dialog Item	Description
Menu Content	text shown in the **Tools** menu. This line may contain key codes and file codes. Shortcuts are defined with a & character. The current selected menu line allows you to specify the options listed below.
Prompt for Arguments	if enabled, a dialog box opens at the time you invoke the menu item that allows you to specify the command line arguments for the user program.
Run Minimized	enable this option to execute the application with minimized window.
Command	program file that is executed with the selected menu item.
Initial Folder	current working folder for the application program. If this entry is empty, µVision2 uses the base folder of the project file.
Arguments	command line arguments that are passed to the application program.

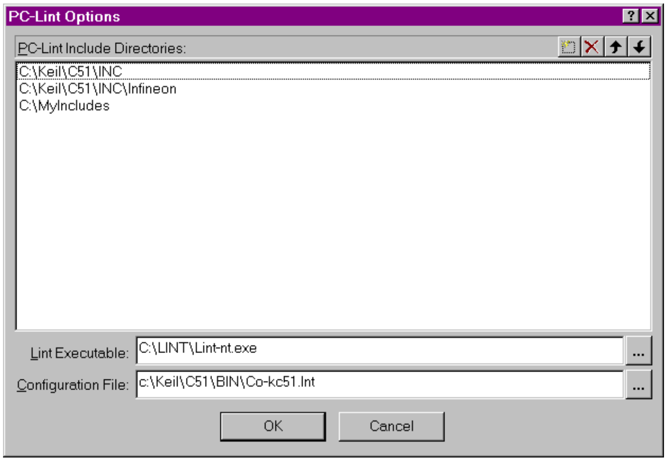
The output of command line based application programs is copied to a temporary file. When the application execution completes the content of this temporary file is listed in the **Output Window – Build** page.

Running PC-Lint

PC-Lint from Gimpel Software checks the syntax and semantics of C programs across all modules of your application. PC-Lint flags possible bugs and inconsistencies and locates unclear, erroneous, or non-sense C code. PC-Lint may considerably reduce the debugging effort of your target application.

Install **PC-Lint** on your PC and enter parameters in the dialog **Tools – Setup PC Lint**. The example shows a typical PC-Lint configuration.

To get correct output in the **Build** page, you should use the configuration file that is located in the folder **KEIL\C51\BIN**.

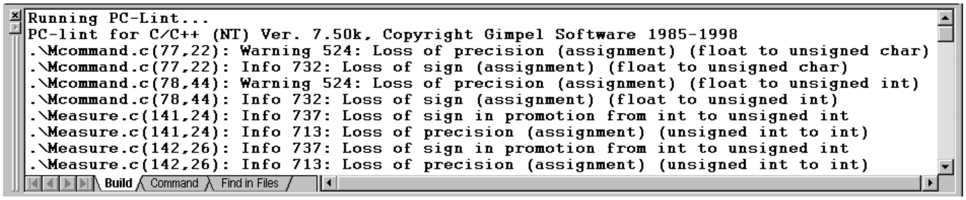


PC-Lint Include Directories:
C:\Keil\C51\INC
C:\Keil\C51\INC\Infineon
C:\MyIncludes

Lint Executable: C:\LINT\Lint-nt.exe

Configuration File: c:\Keil\C51\BIN\Co-kc51.lnt

After the setup of PC-Lint you may *Lint* your source code. **Tools – Lint …** runs PC-Lint on the current in focus editor file. **Tools – Lint All C Source Files** runs PC-Lint across all C source files of your project. The PC-Lint messages are redirected to the **Build – Output Window**. A double click on a Lint message line locates the editor to the source file position.



```
Running PC-Lint...
PC-lint for C/C++ (NT) Ver. 7.50k, Copyright Gimpel Software 1985-1998
.\Mcommand.c(77,22): Warning 524: Loss of precision (assignment) (float to unsigned char)
.\Mcommand.c(77,22): Info 732: Loss of sign (assignment) (float to unsigned char)
.\Mcommand.c(78,44): Warning 524: Loss of precision (assignment) (float to unsigned int)
.\Mcommand.c(78,44): Info 732: Loss of sign (assignment) (float to unsigned int)
.\Measure.c(141,24): Info 737: Loss of sign in promotion from int to unsigned int
.\Measure.c(141,24): Info 713: Loss of precision (assignment) (unsigned int to int)
.\Measure.c(142,26): Info 737: Loss of sign in promotion from int to unsigned int
.\Measure.c(142,26): Info 713: Loss of precision (assignment) (unsigned int to int)
```

To get correct results in the **Build – Output Window**, PC-Lint needs the following option lines in the configuration file:

```
-hsb_3                                      // 3 lines output, column below
-format="*** LINT: %(%f(%l) %)%t %n: %m"    // Change message output format
-width(0,10)                                // Don't break lines
```

The configuration file **C:\KEIL\C51\BIN\CO-KC51.LNT** contains already these lines. It is strongly recommended to use this configuration file, since it contains also other PC-Lint options required for the Keil C51 compiler.

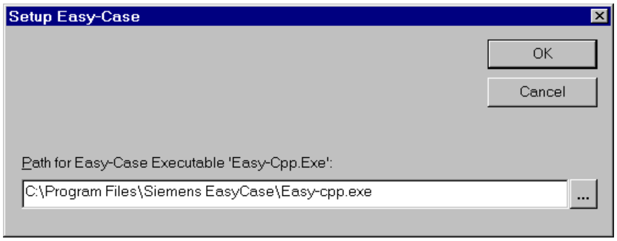
Siemens Easy-Case

µVision2 provides a direct interface to Siemens Easy-Case. EasyCase is a graphic editor as well as a program documentation utility. You may use EasyCase to edit source code. Also some µVision2 debugger commands are available within the EasyCase environment.

Install EasyCase: to use µVision2 debugger commands within Siemens EasyCase the configuration settings from the file **C:\KEIL\UV2\UV2EASY-CPP.INI** should be added to the file **EASY-CPP.INI** that is stored in the **WINDOWS** system directory. This may be done with any text editor or the DOS copy command:

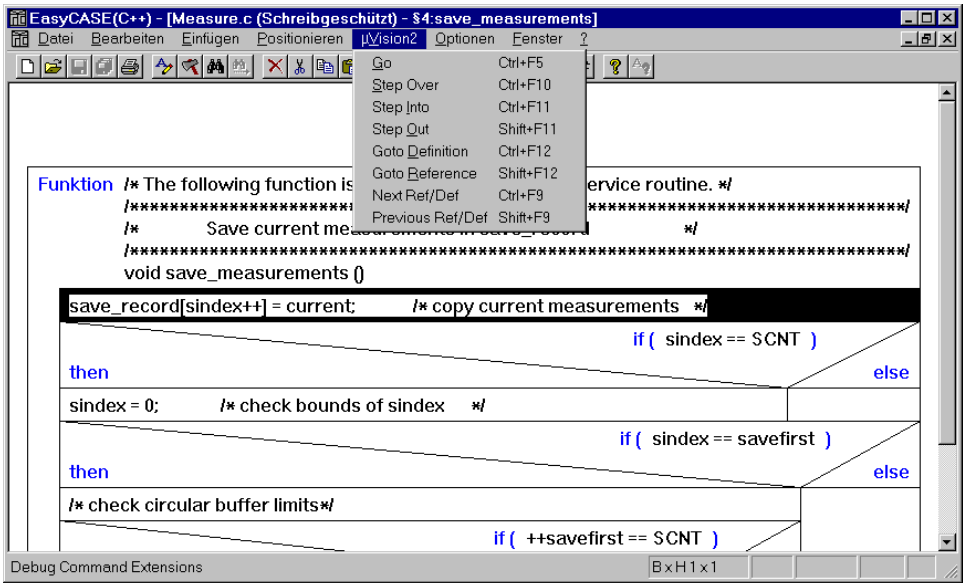
```
C:\>CD C:\WINNT
C:\WINNT>COPY EASY-CPP.INI+C:\KEIL\UV2\UV2EASY-CPP.INI   EASY-CPP.INI
```

In the µVision2 dialog **Tools – Setup Easy-Case** enter the path for **EASY-CPP.EXE.** This completes the configuration for Siemens EasyCase.



Path for Easy-Case Executable 'Easy-Cpp.Exe': C:\Program Files\Siemens EasyCase\Easy-cpp.exe

View Source Code with EasyCase: with **Tools – Start/Stop EasyCase** you start EasyCase. The menu item **Tools – Show …** opens the active µVison2 editor file at the current position. The EasyCase menu **µVision2** offers several debug commands that allow program execution in the µVision2 debugger.

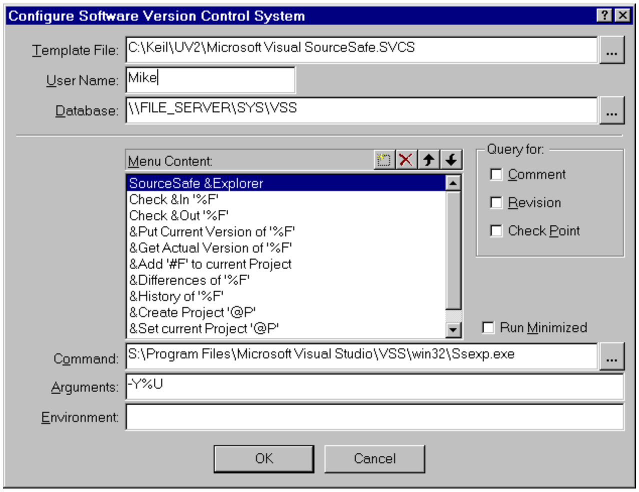


```
/* The following function is                    ervice routine. */
/************************                   ***********************************/
/*            Save current me                                    */
/******************************************************************************/
void save_measurements ()
save_record[sindex++] = current;        /* copy current measurements  */
if (  sindex == SCNT  )
sindex = 0;          /* check bounds of sindex     */
if (  sindex == savefirst  )
/* check circular buffer limits*/
if (  ++savefirst == SCNT  )
```

Using the SVCS Menu

µVision2 provides a configurable interface to Software Version Control Systems (SVCS). Pre-configured template files are provided for: Intersolv PVCS, Microsoft SourceSafe, and MKS Source Integrity.

Via the SVCS Menu you call the command line tools of your Version Control System. The configuration of the **SVCS** menu is stored in a **Template File**. This menu is configured with the dialog **SVCS – Customize SVCS Menu…**. The dialog options are explained below.



Configure Software Version Control System

Template File: C:\Keil\UV2\Microsoft Visual SourceSafe.SVCS

User Name: Mike

Database: \\FILE_SERVER\SYS\VSS

Menu Content:

SourceSafe &Explorer
Check &In '%F'
Check &Out '%F'
&Put Current Version of '%F'
&Get Actual Version of '%F'
&Add '#F' to current Project
&Differences of '%F'
&History of '%F'
&Create Project '@P'
&Set current Project '@P'

Query for:
☐ Comment
☐ Revision
☐ Check Point

☐ Run Minimized

Command: S:\Program Files\Microsoft Visual Studio\VSS\win32\Ssexp.exe

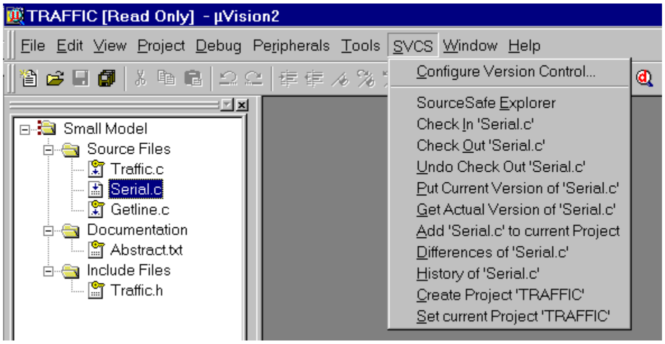
Arguments: -Y%U

Environment:

Dialog Item	Description
Template File	name of the SVCS menu configuration file. It is recommended that all members of the software team are using the same template file. Therefore the template file should be copied to the file server.
User Name	user name that should be used to log into the SVCS system. The user name is passed with the **%U** file code in the argument line.
Database	file name or path for the database used by the SVCS system. The database string is passed with the **%V** file code in the argument line.
Menu Content	text shown in the **SVCS** menu. This line may contain key codes and file codes. Shortcuts are defined with a & character. The selected menu line allows you to specify the options listed below.
Query for … Comment Revision CheckPoint	allows you to ask for additional information when using the SVCS command. A comment is copied into a temporary file that can be passed with the file code **%Q** as argument to the SVCS command. Revision and CheckPoint are passed as a string with **%R** and **%C** file code.
Run Minimized	enable this option to execute the application with minimized window.
Command	program file that is invoked when you click on the SVCS menu item.
Arguments	command line arguments that are passed to the SVCS program file.
Environment	environment variables that are set before execution of the SVCS program.

The output of command line SVCS application programs is copied to a temporary file. When the SVCS command completes the content of this temporary file is listed in the **Output Window – Build** page.

A sample SVCS menu is shown on the right. A selected file in the page **Project Window – Files** is the SVCS argument. The target name selects the ***.UV2** project file. The local copy of a looked file is read-only and gets a key symbol.


TRAFFIC [Read Only] - µVision2
File Edit View Project Debug Peripherals Tools SVCS Window Help
Small Model
Source Files
Traffic.c
Serial.c
Getline.c
Documentation
Abstract.txt
Include Files
Traffic.h
Configure Version Control...
SourceSafe Explorer
Check In 'Serial.c'
Check Out 'Serial.c'
Undo Check Out 'Serial.c'
Put Current Version of 'Serial.c'
Get Actual Version of 'Serial.c'
Add 'Serial.c' to current Project
Differences of 'Serial.c'
History of 'Serial.c'
Create Project 'TRAFFIC'
Set current Project 'TRAFFIC'

µVision2 projects are saved in two separate files. Project settings in ***.UV2:** this file should be looked with the SVCS and is sufficient to re-build an application. The local µVision2 configuration in ***.OPT** contains window positions and debugger settings.

The following table lists typical SVCS menu items. Depending on your configuration, additional or different items might be available. Include files may be added to the project as document file to access them quickly with the SVCS.

SVCS Menu Item	Description
Explorer	start the interactive SVCS explorer.
Check In	save the file in the SVCS database and make the local copy read-only.
Check Out	get the actual file version from the SVCS and allows modifications.
Undo Check Out	undo the check out of a file.
Put Current Version	save a local file in the SVCS database but still allow modifications to it.
Get Actual Version	get a current read-only copy of a file from the SVCS.
Add *file* to Project	add the file to the SVCS project.
Add *file* to Project	add the file to the SVCS project.
Differences, History	show SVCS information about the file.
Create Project	create a SVCS project with the same name as the local µVision2 project.

NOTES

*The pre-configured *.SVCS files may be modified with a text editor to adapt program paths and tool parameters.*

*Microsoft SourceSafe requires the command **Set Current Project** after you have selected a new µVision2 project. Remove the **SSUSER** environment variable from the configuration to use the login name of the workstation.*

MKS Source Integrity is pre-configured to create a project database on a server and a local sandbox workspace on the workstation.

Intersolv PVCS is not pre-configured for creating and maintaining projects.

Writing Optimum Code

Many configuration parameters have influence on the code quality of your 8051 application. Although, for most applications the default tool setting generates very good code, you should be aware of the parameters which improve code density and execution speed. The code optimization techniques are described in this section.

Memory Models and Memory Types

The most significant impact on code size and execution speed has the memory model. The memory model influences variable accesses. Refer to “Memory Models” on page 25 for detailed information. The memory model is selected in the **Options for Target – Target** dialog page.

Global Register Optimization

The Keil 8051 tools provide support for application wide register optimization which is enabled in the Options for Target – C51 dialog with **Global Register Optimization**. With the application wide register optimization, the C51 compiler *knows* the registers used by external functions. Registers that are not altered in external functions can be used to hold register variables. The code generated by the C compiler needs less data and code space and executes faster. To improve the register allocation, the µVision2 build process makes automatically iterative re-translations of C source files.

In the following example *input* and *output* are external functions, which require only a few registers.

With Global Register Optimization	Without Global Register Optimization

main () {
unsigned char i;
unsigned char a;
while (1) {
i = input ();
	&nbsp;
	
	/ get number of values */*

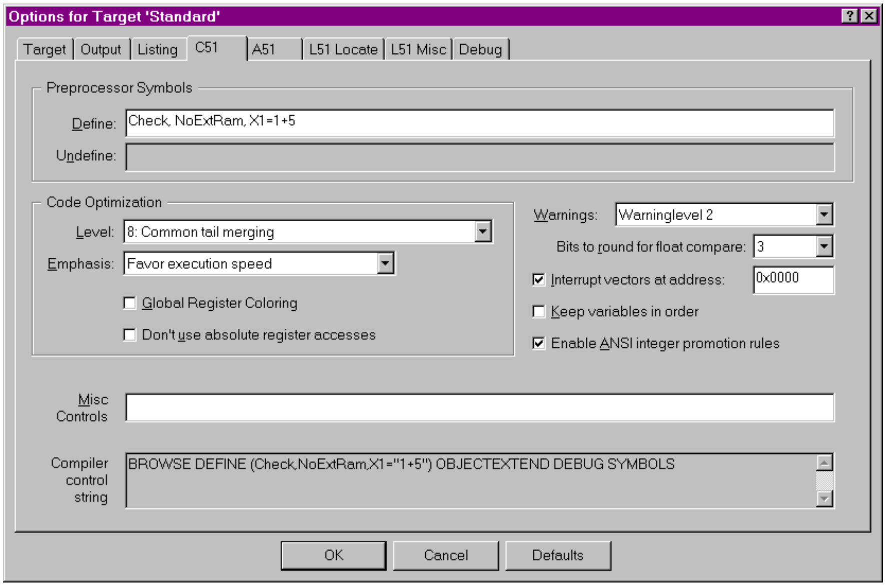
`?C0001:`
`LCALL input`
`;- 'i' assigned to 'R6' -`
`MOV R6,AR7`
	`?C0001:`
	`LCALL input`
	`MOV DPTR,#i`
	`MOV A,R7`
	`MOV @DPTR,A`

do {
a = input ();
	
	/ get input value */*

With Global Register Optimization	Without Global Register Optimization
?C0005: LCALL input ;- 'a' assigned to 'R7' - MOV R5,AR7	?C0005: LCALL input MOV DPTR,#a MOV A,R7 MOVX @DPTR,A
output (a);	*/* output value */*
LCALL _output	LCALL _output
} while (--i);	*/* decrement values */*
DJNZ R6,?C0005	MOV DPTR,#i MOVX A,@DPTR DEC A MOVX @DPTR,A JNZ ?C0005
}	
SJMP ?C0001	SJMP ?C0001
}	
RET	RET
Code Size: 18 Bytes	**Code Size: 30 Bytes**

Other C51 Compiler Directives

There are several other C51 directives that improve the code quality. These directives are enabled in the Options – C51 dialog page. You can translate the C modules in an application with different compiler settings. You may check the code quality of different compiler settings in the listing file.



Options for Target 'Standard'

Target | Output | Listing | C51 | A51 | L51 Locate | L51 Misc | Debug

Preprocessor Symbols

Define: Check, NoExtRam, X1=1+5

Undefine:

Code Optimization

Level: 8: Common tail merging

Emphasis: Favor execution speed

☐ Global Register Coloring

☐ Don't use absolute register accesses

Warnings: Warninglevel 2

Bits to round for float compare: 3

☑ Interrupt vectors at address: 0x0000

☐ Keep variables in order

☑ Enable ANSI integer promotion rules

Misc Controls:

Compiler control string: BROWSE DEFINE (Check,NoExtRam,X1="1+5") OBJECTEXTEND DEBUG SYMBOLS

The following table describes the options of the **C51** dialog page:

Dialog Item	Description
Define	outputs the C51 **DEFINE** directive to enter preprocessor symbols.
Undefine	is only available in the **Group** and **File Options** dialog and allows you to remove DEFINE symbols that are specified at the higher target or group level.
Code Optimization Level	specifies C51 OPTIMIZE level. Typical you will not alter the default. With the highest level "9: Common block subroutine packing" the compiler detects multiple instruction sequences and packs such code into subroutines. While analyzing the code, the compiler also tries to replace sequences with cheaper instructions. Since the compiler inserts sub-routines and CALL instructions, the execution speed of the optimized code might be slower. Typical this level is interesting to optimize the code density.
Code Optimization Emphasis	You can optimize for execution speed or code size. With "Favor Code Size", the C51 compiler uses library calls instead of fast replacement code.
Global Register Optimization	enables the "Global Register Optimization". Refer to page 67 for details.
Don't use absolute register accesses	disables absolute register addressing for registers R0 through R7. The code will be slightly longer, since C51 cannot use *ARx* symbols, i.e. in PUSH or POP instructions and needs to insert replace code. However the code will be independent of the selected register bank.
Warnings	selects the C51 warninglevel. Warninglevel 0 disables all warnings.
Bits to round for float compare	determines the number of bits to rounded before a floating-point compare is executed.
Interrupt vectors at address	instructs the C51 compiler to generate interrupt vectors for interrupt functions and specifies the base address for the interrupt vector table.
Keep Variables in Order	tells the C51 compiler to order the variables in memory according their definition in the C source file. This option does not influence code quality.
Enable ANSI interger promotion rules	expressions used in if statements are promoted from smaller types to integer expressions before comparison. This gives typically longer code, but is required to be ANSI compatible.
Misc Controls	allows you to enter special C51 directives. You may need such options when you are using very new 8051 devices that require special directives.
Compiler Control String	displays the C51 compiler invocation string. Allows you can verify the compiler options currently for your source files.

Data Types

The 8051 CPU is an 8-bit microcontroller. Operations that use 8-bit types (like **char** and **unsigned char**) are more efficient than operations that use int or long types.

Tips and Tricks

The following section discusses advanced techniques you may use with the µVision2 project manager. You will not need the following features very often, but readers of this section get a better feeling for the µVision2 capabilities.

Import Project Files from µVision Version 1

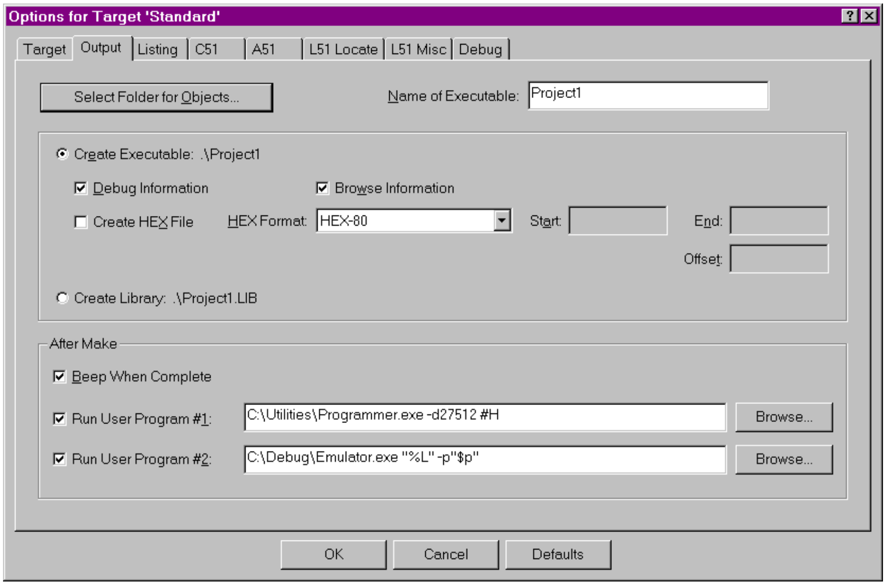
You can import project files from µVision1 with the following procedure:

1. Create a new µVision2 project file and select a CPU from the device database as described on page 48. It is important to create the new µVision2 project file in the existing µVision1 project folder.
2. Select the old µVision1 project file that exists in the project folder in the dialog **Project – Import µVision1 Project**. This menu option is only available, if the file list of the new **µVision2** project file is empty.
3. This command imports the old µVision1 linker settings into the linker dialogs. But, we recommend that you are using the µVision2 **Options for Target – Target** dialog to define the memory structure of your target hardware. Once you have done that, you should enable **Use Memory Layout from Target Dialog** in the **Options for Target – L51 Locate** dialog and remove the settings for **User Classes** and **User Sections** in this dialog.
4. Check carefully if all settings are copied correctly to the new µVision2 project file.
5. You may now create file groups in the new µVision2 project as described under “Project Targets and File Groups” on page 54. Then you can **Drag & Drop** files into the new file groups.

NOTE
It is not possible to make a 100% conversion from µVision1 project files since µVision2 differs in many aspects from the previous version. After you have imported your µVision1 check carefully if the tool settings are converted correctly. Some µVision1 project settings, for example user translator and library module lists are not converted to the µVision2 project. Also the dScope Debugger settings cannot be copied to the µVision2 project file.

Start External Tools after the Build Process

The **Options for Target – Output** dialog allows to enter up to two users programs that are started after a successful build process. Using a key sequence you may pass arguments from the µVision2 project manager to these user programs. Refer to "Key Sequence for Tool Parameters" on page 60.



Name of Executable: Project1

● Create Executable: .\Project1

☑ Debug Information ☑ Browse Information

□ Create HEX File HEX Format: HEX-80 Start: End:

Offset:

○ Create Library: .\Project1.LIB

After Make

☑ Beep When Complete

☑ Run User Program #1: C:\Utilities\Programmer.exe -d27512 #H

☑ Run User Program #2: C:\Debug\Emulator.exe "%L" -p"$p"

In the example above the **User Program #1** is called with the Hex Output file and the full path specification i.e. **C:\MYPROJECT\PROJECT1.HEX**. The **User program #2** will get only the name of the linker output file **PROJECT1** and as a parameter ***-p*** the path specification to the project **C:\MYPROJECT**. You should enclose the key sequence with quotes ("") if you use folder names with special characters, i.e. space, ~, #.

Specify a Separate Folder for Listing and Object Files

You can direct the output files of the tools to different folders:

- The **Options for Target – Output** dialog lets you **Select a Folder for Objects**. When you use a separate folder for the object files of each project target, µVision2 has still valid object files of the previous build process. Even when you change your project target, a **Build Target** command will just re-translate the modified files.
- The **Options for Target – Listing** dialog provides the same functionality for all listing files with the **Select Folder for List Files** button.

Use a CPU that is not in the µVision2 Device Database

The µVision2 device database contains all 8051 standard products. However, there are some custom devices and there will be future devices which are currently not part of this database. If you need to work with an unlisted CPU you have two alternatives:

- Select a device listed under the rubric **Generic**. The **8051 (all Variants)** device allows you to configure all tool parameters and therefore supports all CPU variants. Specify the on-chip memory as External Memory in the **Options for Target – Target** dialog.
- You may enter a new CPU into the µVision2 device database. Open the dialog **File – Device Database** and select a CPU that comes close to the device you want to use and modify the parameters. The **CPU** setting in the **Options** box defines the basic the tool settings. The parameters are described in the following table.

Parameter	Specifies …
IRAM (*range*)	Address location of the on-chip IRAM.
XRAM (*range*)	Address location of the on-chip XRAM.
IROM (*range*)	Address range of the on-chip (flash) ROM. The start address must be 0.
CLOCK (*val*)	Default CPU clock used when you select the device.
MODA2	Dual DPTR for Atmel device variants.
MODDP2	Dual DPTR for Dallas device variants.
MODP2	Dual DPTR for Philips and Temic device variants.
MOD517DP	Multiple DPTR for Infineon C500 device variants.
MOD517AU	Arithmetic Unit for Infineon C500 device variants.

Other **Option** variables specify CPU data books and µVision2 Debugging DLLs.
Leave this variables unchanged when adding a new device to the database.

Create a Library File

Select **Create Library** in the dialog **Options for Target – Output**. µVision2 will call the Library Manager instead of the Linker/Locater. Since the code in the Library will be not linked and located, the entries in the **L51 Locate** and **L51 Misc** options page are ignored. Also the CPU and memory settings in the **Target** page are not relevant. Select a CPU listed under the rubric **Generic** in the device database, if you plan to use your code on different 8051 directives.

Copy Tool Settings to a New Target

Select **Copy all Settings from Current Target** when you add a new target in the **Project – Targets, Groups, Files…** dialog. Copy tool settings from an existing target to the current target in following way:

1. Use **Remove Target** to delete the current target.
2. Select the target with the tool settings you want to copy with **Set as Current Target**.
3. **Add** the again removed target with **Copy all Settings from Current Target** enabled.

Locate Segments to Absolute Memory Locations

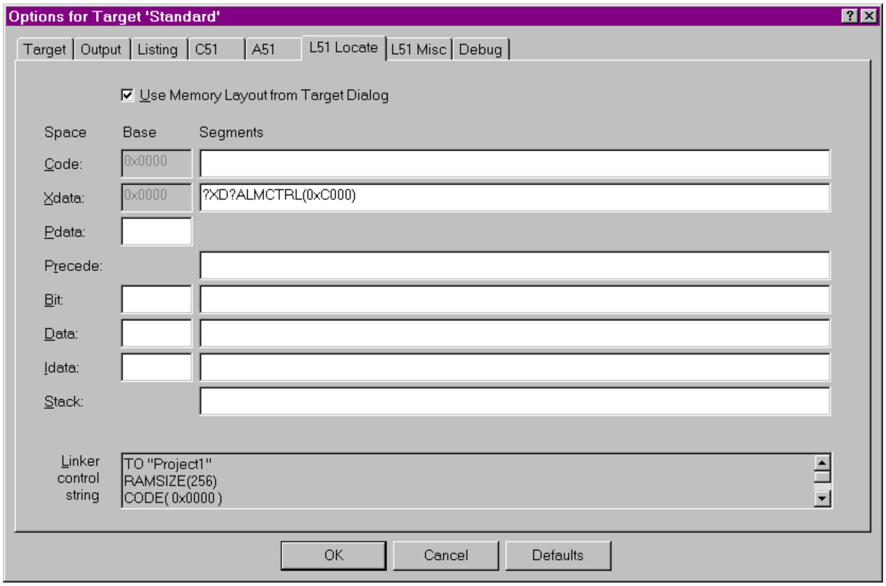
Sometimes, it is required to locate sections to specific memory addresses. In the following example, the structure called **`alarm_control`** should be located at address 0xC000. This structure is defined in a source file named **ALMCTRL.C** and this module contains only the declaration for this structure.

```
:
:
struct alarm_st  {
  unsigned int alarm_number;
  unsigned char enable flag;
  unsigned int time_delay;
  unsigned char status;
};

struct alarm_st xdata alarm_control;
:
:
```

The C51 compiler generates an object file for **ALMCTRL.C** and includes a segment for variables in the **xdata** memory area. The variable **`alarm_control`** is the located in the segment **?XD?ALMCTRL**. µVision2 allows you to specify the base address of any section under **Options for Target – L51 Locate – Users Sections**. In the following example linker/locater will locate the section named **?XD?ALMCTRL** at address 0xC000 in the physical xdata memory.

NOTE
C51 offers you also _at_ directive and absolute memory access macros to address absolute memory locations. For more information refer to the “C51 User's Guide”, Chapter 6.



Options for Target 'Standard'

☑ Use Memory Layout from Target Dialog

Space	Base	Segments
Code:	0x0000	
Xdata:	0x0000	?XD?ALMCTRL(0xC000)
Pdata:		
Precede:		
Bit:		
Data:		
Idata:		
Stack:		

Linker control string:

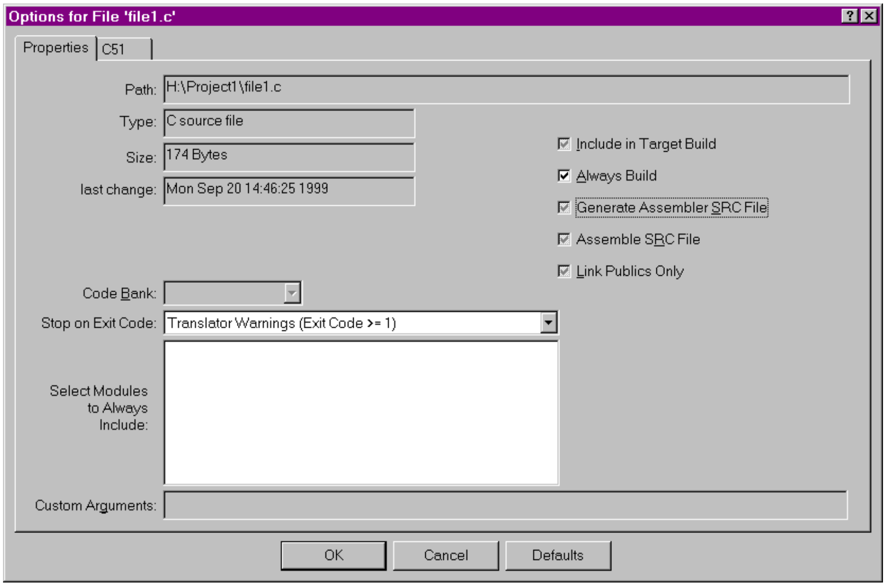
```
TO "Project1"
RAMSIZE(256)
CODE( 0x0000 )
```

File and Group Specific Options – Properties Dialog

µVision2 allows you to set file and group specific options via the local menu in the **Project Window – Files** page as follows: select a file or group, click with the right mouse key and choose **Options for …**. Then you can review information or set special options for the item selected. The dialog pages have tri-state controls. If a selection is gray or contains <*default*> the setting from the higher group or target level is active. The following table describes the options of the **Properties** dialog page:

Dialog Item	Description
Path, Type, Size Last Change	Outputs information about the file selected.
Include in Target Build	Disable this option to exclude the group or source file in this target. If this option is not set, µVision2 will not translate and not link the selected item into the current targets. This is useful for configuration files, when you are using the project file for several different hardware systems.
Always Build	Enable this option to re-translate a source module with every build process, regardless of modifications in the source file. This is useful when a file contains **__DATE__** and **__TIME__** macros that are used to stored version information in the application program.
Generate Assembler SRC File	Instructs the C51 compiler to generate an assembler source file from this C module. Typical this option is used when the C source file contains **#pragma asm / endasm** sections.
Assemble SRC File	Use this option together with the **Generate Assembler SRC File** to translate the assembler source code generated by C166 into an object file that can be linked to the application.

Dialog Item	Description
Link Publics Only	This option is only available with L251 and instructs the linker to include only PUBLIC symbols from that module. Typical this option when you want to use entry or variable addresses from a different application. It refers in the most cases to an absolute object file which may be part of the project.
Stop on Exit Code	Specify an exit code when the build process should be stop on translator messages. By default, µVision2 translates all files in a build process regardless of error or warning messages.
Select Modules to Always Include	Allows you to always include specific modules from a Library. Refer to “Include Always specific Library Modules” on page 77 for more information.
Custom Arguments	This line is required if your project contains files that need a different translator. Refer to “Use a Custom Translator” on page 78 for more information.



Options for File 'file1.c'

Path: H:\Project1\file1.c

Type: C source file

Size: 174 Bytes

last change: Mon Sep 20 14:46:25 1999

☑ Include in Target Build

☑ Always Build

☑ Generate Assembler SRC File

☑ Assemble SRC File

☑ Link Publics Only

Code Bank:

Stop on Exit Code: Translator Warnings (Exit Code >= 1)

Select Modules to Always Include:

Custom Arguments:

In this example we have specified for **FILE1.C** that the build process is stopped when there are translator warnings and that this file is translated with each build process regardless of modifications.

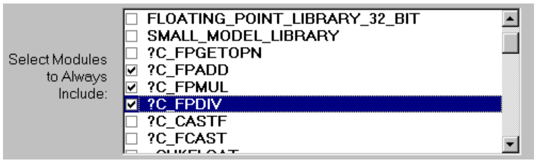
Translate a C Module with asm / endasm Sections

If you use within your C source module assembler statements, the C51 compiler requires you to generate an assembler source file and translate this assembler source file. In this case enable the options **Generate Assembler SRC File** and **Assembler SRC File** in the properties dialog.

NOTE
Check if you can use build-in intrinsic functions to replace the assembler code. In general it better to avoid assembler code sections since you C source code will not be portable to other platforms. The C51 compiler offers you many intrinsic functions that allow you to access all special peripherals. Typically it is not required to insert assembler instructions into C source code.

Include Always specific Library Modules

The Properties dialog page allows you to specify library modules that should be always included in a project. This is sometimes required when you generate a boot portion of an application that should contain generic routines that are used from program parts that are reloaded later. In this case add the library that contains the desired object modules, open the **Options – Properties** dialog via the local menu in the **Project Window – Files** and **Select Modules to Always Include**.



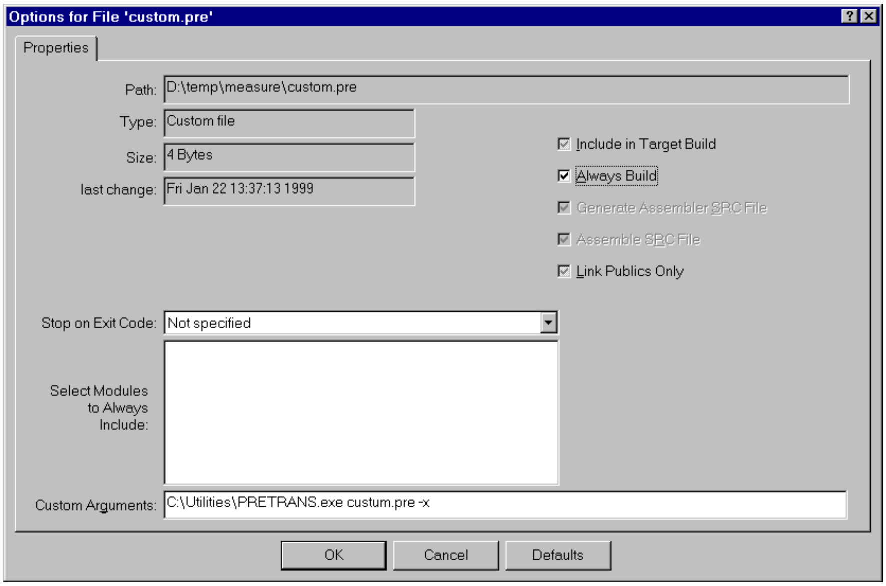
Select Modules to Always Include:

- ☐ FLOATING_POINT_LIBRARY_32_BIT
- ☐ SMALL_MODEL_LIBRARY
- ☐ ?C_FPGETOPN
- ☑ ?C_FPADD
- ☑ ?C_FPMUL
- ☑ ?C_FPDIV
- ☐ ?C_CASTF
- ☐ ?C_FCAST

Just enable the modules you want to include in any case into your target application.

Use a Custom Translator

If you add a file with unknown file extension to a project, µVision2 requires you to specify the file type for this file. You may select **Custom File** and use a custom translator to process this file. The custom translator is specified along with its command line in the **Custom Arguments** line of the **Options – Properties** dialog. Typical the custom translator will generate a source file from the custom file. You need to add this source file to your project to and use A51 or C51 to generate an object file that can be linked to your application.



Options for File 'custom.pre'

Properties

Path: D:\temp\measure\custom.pre

Type: Custom file

Size: 4 Bytes

last change: Fri Jan 22 13:37:13 1999

☑ Include in Target Build

☑ Always Build

☑ Generate Assembler SRC File

☑ Assemble SRC File

☑ Link Publics Only

Stop on Exit Code: Not specified

Select Modules to Always Include:

Custom Arguments: C:\Utilities\PRETRANS.exe custum.pre -x

In this example we have specified for **CUSTOM.PRE** that the program **C:\UTILITIES\PRETRANS.EXE** is used with the parameter **–X** to translate the file. Note that we have used also the **Always Build** option to ensure that the file is translated with every build process.

File Extensions

The dialog **Project – File Extensions** allows you to set the default file extension for a project. You can enter several extensions when you separate them with semi-colon characters. The file extensions are project specific.

Different Compiler and Assembler Settings

Via the local menu in the **Project Window – Files** you may set different options for a file group or even a single file. The dialog pages have tri-state controls; if an option is grayed the setting from higher level is taken. You can specify with this technique different tools for a complete file group and still change settings on a single source file within this file group.

Version and Serial Number Information

Detailed tool chain information is listed when you open **Help – About**. Please use this information whenever you send us problem reports.

Chapter 5. Testing Programs

µVision2 Debugger

You can use µVision2 Debugger to test the applications you develop using the C51 compiler and A51 macro assembler. The µVision2 Debugger offers two operating modes that are selected in the **Options for Target – Debug** dialog:

Use Simulator allows to configure the µVision2 Debugger as *software-only product* that simulates most features of the 8051 microcontroller family without actually having target hardware. You can test and debug your embedded application before the hardware is ready. µVision2 simulates a wide variety of peripherals including the serial port, external I/O, and timers. The peripheral set is selected when you select a CPU from the device database for your target.

Use Advance GDI drivers, like **Keil Monitor 51** interface. With the Advanced GDI interface you may connect the environment directly to emulators or the Keil Monitor program. For more information refer to “Chapter 11. Using Monitor-51” on page 175.

CPU Simulation

µVision2 simulates up to 16 Mbytes of memory from which areas can be mapped for read, write, or code execution access. The µVision2 simulator traps and reports illegal memory accesses.

In addition to memory mapping, the simulator also provides support for the integrated peripherals of the various 8051 derivatives. The on-chip peripherals of the CPU you have selected are configured from the Device Database selection you have made when you create your project target. Refer to page 48 for more information about selecting a device.

You may select and display the on-chip peripheral components using the **Debug** menu. You can also change the aspects of each peripheral using the controls in the dialog boxes.

Start Debugging

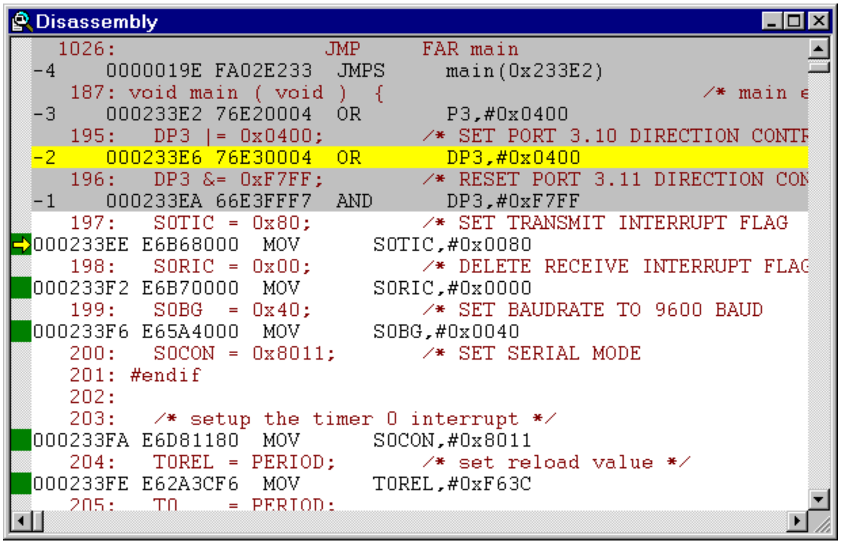
You start the debug mode of µVision2 with the **Debug – Start/Stop Debug Session** command. Depending on the **Options for Target – Debug** configuration, µVision2 will load the application program and run the startup code. For information about the configuration of the µVision2 debugger refer to page 88. µVision2 saves the editor screen layout and restores the screen layout of the last debug session. If the program execution stops, µVision2 opens an editor window with the source text or shows CPU instructions in the disassembly window. The next executable statement is marked with a yellow arrow.

During debugging, most editor features are still available. For example, you can use the find command or correct program errors. Program source text of your application is shown in the same windows. The µVision2 debug mode differs from the edit mode in the following aspects:

- The "Debug Menu and Debug Commands" described on page 19 are available. The additional debug windows are discussed in the following.
- The project structure or tool parameters cannot be modified. All build commands are disabled.

Disassembly Window

The Disassembly window shows your target program as mixed source and assembly program or just assembly code. A trace history of previously executed instructions may be displayed with **Debug – View Trace Records**. To enable the trace history, set **Debug – Enable/Disable Trace Recording**.



```
 1026:                  JMP      FAR main
-4     0000019E FA02E233  JMPS     main(0x233E2)
  187: void main ( void )  {                              /* main e
-3     000233E2 76E20004  OR       P3,#0x0400
  195:    DP3 |= 0x0400;        /* SET PORT 3.10 DIRECTION CONTR
-2     000233E6 76E30004  OR       DP3,#0x0400
  196:    DP3 &= 0xF7FF;        /* RESET PORT 3.11 DIRECTION CON
-1     000233EA 66E3FFF7  AND      DP3,#0xF7FF
  197:    S0TIC = 0x80;         /* SET TRANSMIT INTERRUPT FLAG
000233EE E6B68000  MOV      S0TIC,#0x0080
  198:    S0RIC = 0x00;         /* DELETE RECEIVE INTERRUPT FLAG
000233F2 E6B70000  MOV      S0RIC,#0x0000
  199:    S0BG  = 0x40;         /* SET BAUDRATE TO 9600 BAUD
000233F6 E65A4000  MOV      S0BG,#0x0040
  200:    S0CON = 0x8011;       /* SET SERIAL MODE
  201: #endif
  202:
  203:    /* setup the timer 0 interrupt */
000233FA E6D81180  MOV      S0CON,#0x8011
  204:    T0REL = PERIOD;       /* set reload value */
000233FE E62A3CF6  MOV      T0REL,#0xF63C
  205:    T0     = PERIOD;
```

If you select the Disassembly Window as the active window all program step commands work on CPU instruction level rather than program source lines. You can select a text line and set or modify code breakpoints using toolbar buttons or the context menu commands.

You may use the dialog **Debug – Inline Assembly…** to modify the CPU instructions. That allows you to correct mistakes or to make temporary changes to the target program you are debugging.

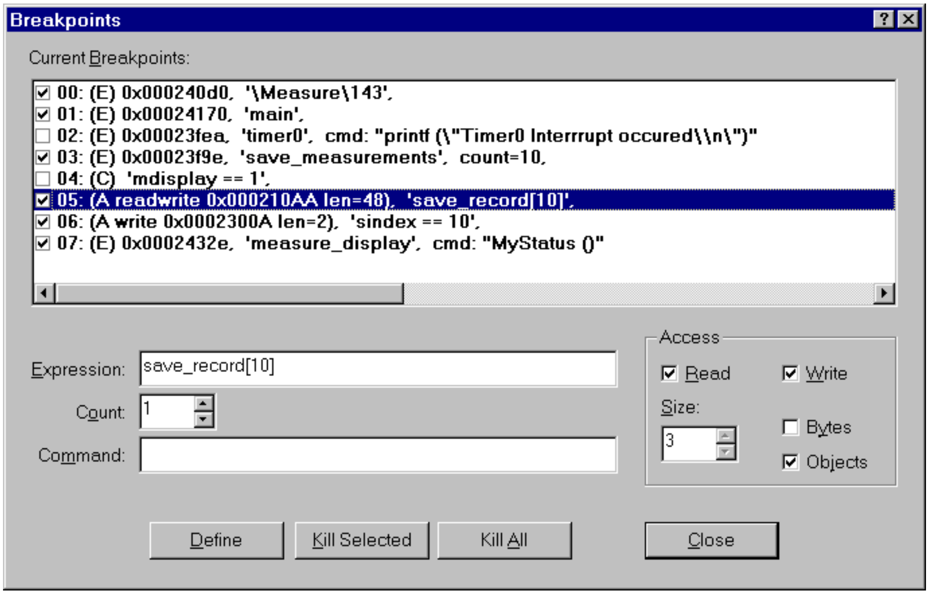
Breakpoints

µVision2 lets you define breakpoints in several different ways. You may already set **Execution Breaks** during editing of your source text, even before the program code is translated. Breakpoints can be defined and modified in the following ways:

- With the **File Toolbar** buttons. Just select the code line in the **Editor** or **Disassembly** window and click on the breakpoint buttons.
- With the breakpoint commands in the local menu. The local menu opens with a right mouse click on the code line in the **Editor** or **Disassembly** window.
- The **Debug – Breakpoints…** dialog lets you review, define and modify breakpoint settings. This dialog allows you to define also access breakpoints with different attributes. Refer to the examples below.

- In the **Output Window – Command** page you can use the **BreakSet**, **BreakKill**, **BreakList**, **BreakEnable**, and **BreakDisable** commands.

The **Breakpoint** dialog lets you view and modify breakpoints. You can quickly disable or enable the breakpoints with a mouse click on the check box in the **Current Breakpoints** list. A double click in the **Current Breakpoints** list allows you to modify the selected break definition.



Current Breakpoints:

```
☑ 00: (E) 0x000240d0, '\Measure\143',
☑ 01: (E) 0x00024170, 'main',
☐ 02: (E) 0x00023fea, 'timer0', cmd: "printf (\"Timer0 Interrrupt occured\\n\")"
☑ 03: (E) 0x00023f9e, 'save_measurements', count=10,
☐ 04: (C) 'mdisplay == 1',
☑ 05: (A readwrite 0x000210AA len=48), 'save_record[10]',
☑ 06: (A write 0x0002300A len=2), 'sindex == 10',
☑ 07: (E) 0x0002432e, 'measure_display', cmd: "MyStatus ()"
```

Expression: save_record[10]

Count: 1

Command:

Access: ☑ Read ☑ Write

Size: 3 ☐ Bytes ☑ Objects

You define a breakpoint by entering an **Expression** in the Breakpoint dialog. Depending on the expression one of the following breakpoint types is defined:

- When the expression is a code address, an **Execution Break (E)** is defined that becomes active when the specified code address is reached. The code address must refer to the first byte of a CPU instruction.

- When a memory **Access** (Read, Write or both) is selected an **Access Break (A)** is defined that becomes active when the specified memory access occurs. You can specify the size of the memory access window in bytes or object size of the expression. Expressions for an **Access Break** must reduce to a memory address and memory type. The operators (&, &&, <. <=. >, >=, = =, and !=) can be used to compare the variable values before the **Access Break** halts program execution or executes the **Command**.

- When the expression cannot be reduced to an address a **Conditional Break (C)** is defined that becomes active when the specified conditional expression becomes true. The conditional expression is recalculated after each CPU instruction, therefore the program execution speed may slow down considerably.

When you specify a **Command** for a breakpoint, µVision2 executes the command and resumes executing your target program. The command you specify here may be a µVision2 debug or signal function. To halt program execution in a µVision2 function, set the **_break_** system variable. For more information refer to “System Variables” on page 101.

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

Breakpoint Examples:

The following description explains the definitions in the Breakpoint dialog shown above. The **Current Breakpoints** list summarizes the breakpoint type and the physical address along with the **Expression**, **Command** and **Count**.

```
Expression:  \Measure\143
```

Execution Break (E) that halts when the target program reaches the code line 143 in the module MEASURE.

```
Expression:  main
```

Execution Break (E) that halts when the target program reaches the **main** function.

```
Expression:  timer0   Command:  printf ("Timer0 Interrupt occurred\n")
```

Execution Break (E) that prints the text "Timer0 Interrupt occurred" in the **Output Window – Command** page when the target program reaches the **timer0** function. This breakpoint is disable in the above Breakpoint dialog.

```
Expression:  save_measurements   Count:  10
```

Execution Break (E) that halts when the target program reaches the function **save_measurements** the 10th time.

```
Expression:  mcommand == 1
```

Contional Break (C) that halts program execution when the expression **mcommand = = 1** becomes true. This breakpoint is disable in the above Breakpoint dialog.

```
Expression:  save_record[10]   Access:  Read Write   Size:  3 Objects
```

Access Break (A) that halts program execution when an read or write access occurs to save_record[10] and the following 2 objects. Since save_record is a structure with size 16 bytes this break defines an access region of 48 bytes.

```
Expression:  sindex == 10       Access:  Write
```

Access Break (A) that halts program execution when the value 10 is written to the variable sindex.

```
Expression:  measure_display    Command:  MyStatus ()
```

Execution Break (E) that executes the µVision2 debug function MyStatus when the target program reaches the function **measure_display**. The target program execution resumes after the debug function MyStatus has been executed.

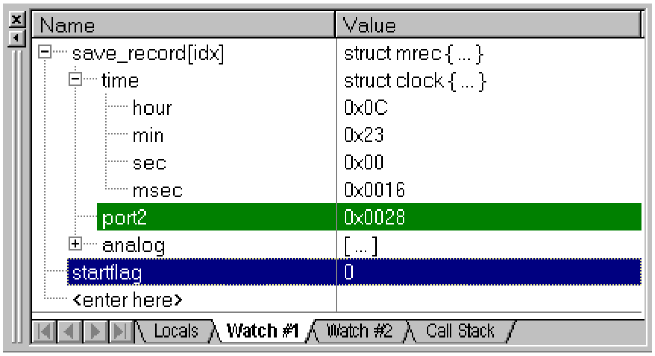
Target Program Execution

µVision2 lets execute your application program in several different ways:

- With the **Debug Toolbar** buttons and the "Debug Menu and Debug Commands" as described on page 19.
- With the **Run till Cursor line** command in the local menu. The local menu opens with a right mouse click on the code line in the **Editor** or **Disassembly** window.
- In the **Output Window – Command** page you can use the **Go**, **Ostep**, **Pstep**, and **Tstep** commands.

Watch Window

The Watch window lets you view and modify program variables and lists the current function call nesting. The contents of the Watch Window are automatically updated whenever program execution stops. You can enable **View – Periodic Window Update** to update variable values while a target program is running.



Name	Value
save_record[idx]	struct mrec { ... }
time	struct clock { ... }
hour	0x0C
min	0x23
sec	0x00
msec	0x0016
port2	0x0028
analog	[...]
startflag	0
<enter here>	

The **Locals** page shows all local function variables of the current function. The **Watch** pages display user-specify program variables. You add variables in three different ways:

- Select the text **<enter here>** with a mouse click and wait a second. Another mouse click enters edit mode that allows you to add variables. In the same way you can modify variable values.
- In an editor window open the context menu with a right mouse click and use **Add to Watch Window**. µVision2 automatically selects the variable name under the cursor position, alternatively you may mark an expression before using that command.
- In the **Output Window – Command** page you can use the **WatchSet** command to enter variable names.

To remove a variable, click on the line and press the **Delete** key.

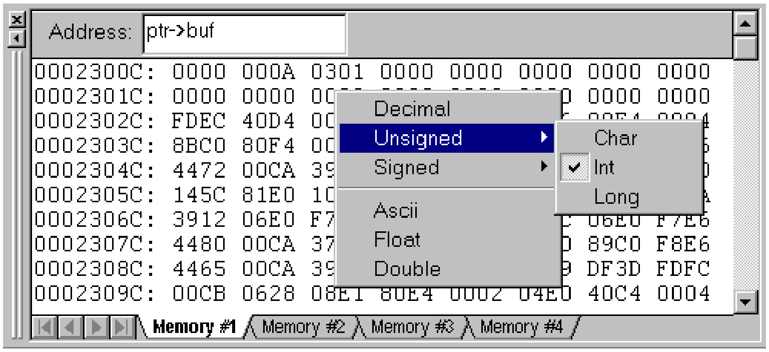
The current function call nesting is shown in the **Call Stack** page. You can double click on a line to show the invocation an editor window.

CPU Registers

The CPU registers are displayed and **Project Window – Regs** page and can be modified in the same way as variables in the **Watch Window**.

Memory Window

The Memory window displays the contents of the various memory areas. Up to four different areas can be review in the different pages. The context menu allows you to select the output format.



Address: ptr->buf

```
0002300C: 0000 000A 0301 0000 0000 0000 0000 0000
0002301C: 0000 0000 00                  0 0000 0000
0002302C: FDEC 40D4 00
0002303C: 8BC0 80F4 00
0002304C: 4472 00CA 39
0002305C: 145C 81E0 10
0002306C: 3912 06E0 F7
0002307C: 4480 00CA 37                  0 89C0 F8E6
0002308C: 4465 00CA 39                  9 DF3D FDFC
0002309C: 00CB 0628 08E1 80E4 0002 04E0 40C4 0004
```

In the **Address** field of the Memory Window, you can enter any expression that evaluates to a start address of the area you want to display. To change the memory contents, double click on a value. This opens an edit box that allows you to enter new memory values. To update the memory window while a target program is running enable **View – Periodic Window Update**.

Toolbox

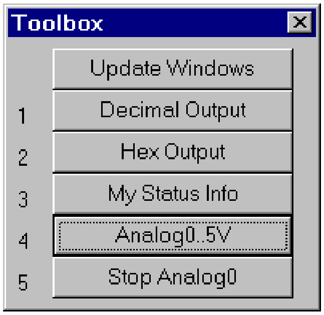
The **Toolbox** contains user-configurable buttons. Click on a **Toolbox** button to execute the associated command. Toolbox buttons may be executed at any time, even while running the test program.

Toolbox buttons are define the **Output Window – Command** page with the DEFINE BUTTON command. The general syntax is:

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

button_label is the name to display on the button in the Toolbox.

command is the µVision2 command to execute when the button is pressed.



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

```
>DEFINE BUTTON "Decimal Output", "radix=0x0A"
>DEFINE BUTTON "Hex Output", "radix=0x10"
>DEFINE BUTTON "My Status Info", "MyStatus ()"   /* call debug function  */
>DEFINE BUTTON "Analog0..5V", "analog0 ()"       /* call signal function */
>DEFINE BUTTON "Show R15", "printf (\"R15=%04XH\\n\")"
```

NOTE
The printf command defined in the last button definition shown above introduces nested strings. The double quote (") and backslash (\) characters of the format string must be escaped with \ to avoid syntax errors.

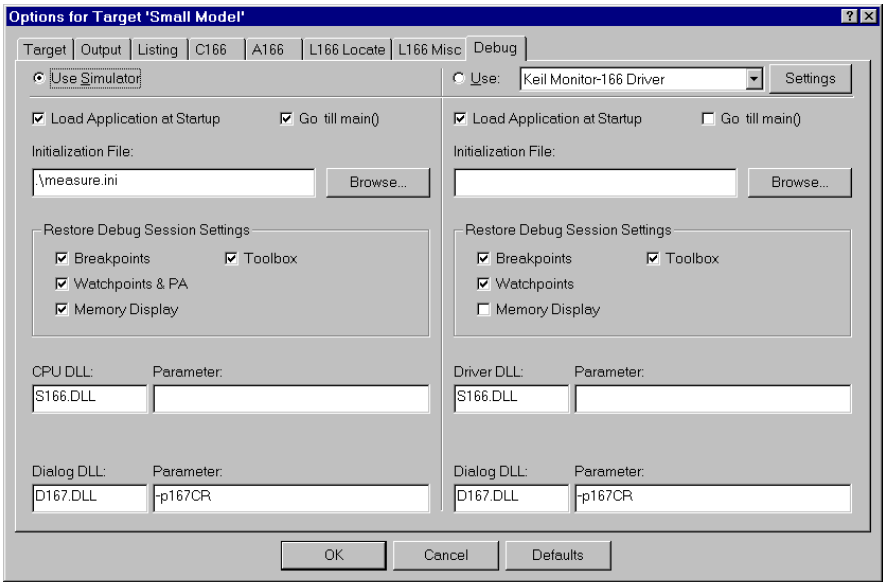
You may remove a **Toolbox** button with the **KILL BUTTON** command and the button number. For example:

```
>Kill Button 5      /* Remove Show R15 button */
```

NOTE
The ***Update Windows*** *button in the Toolbox is created automatically and cannot be removed. The* ***Update Windows*** *button updates several debug windows during program execution.*

Set Debug Options

The dialog **Options for Target - Debug** configures the µVision2 debugger.



Options for Target 'Small Model'

Target | Output | Listing | C166 | A166 | L166 Locate | L166 Misc | Debug

● Use Simulator

☑ Load Application at Startup ☑ Go till main()

Initialization File: .\measure.ini Browse...

Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox
☑ Watchpoints & PA
☑ Memory Display

CPU DLL: S166.DLL Parameter:

Dialog DLL: D167.DLL Parameter: -p167CR

○ Use: Keil Monitor-166 Driver Settings

☑ Load Application at Startup ☐ Go till main()

Initialization File: Browse...

Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox
☑ Watchpoints
☐ Memory Display

Driver DLL: S166.DLL Parameter:

Dialog DLL: D167.DLL Parameter: -p167CR

OK Cancel Defaults

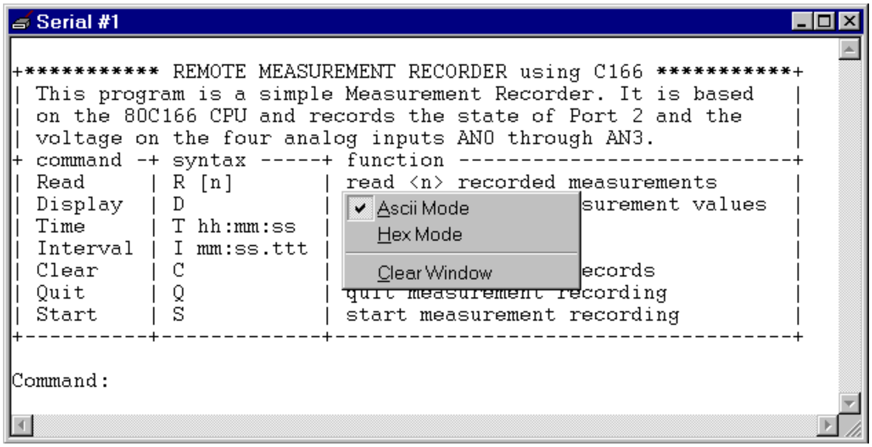
The following table describes the options of the **Debug** dialog page:

Dialog Item	Description
Use Simulator	Select the µVision2 Simulator as Debug engine.

Dialog Item	Description
Use Keil Monitor-166 Driver	Select the Advanced GDI driver to connect to your debug hardware. The Keil Monitor-166 Driver allows you to connect a target board with the Keil Monitor. There are µVision2 emulator and OCDS drivers in preparation.
Settings	Opens the configuration dialog of the selected Advanced GDI driver.
Other dialog options are available separately for the Simulator and Advanced GDI section.	
Load Application at Startup	Enable this option to load your target application automatically when you start the µVision2 debugger.
Go till main ()	Start program execution till the main label when you start the debugger.
Initialization File	Process the specified file as command input when starting a debug session.
Breakpoints	Restore breakpoint settings from the previous debug session.
Toolbox	Restore toolbox buttons from the previous debug session.
Watchpoints & PA	Restore Watchpoint and Performance Analyzer settings from the previous debug session.
Memory Display	Restore the memory display settings from the previous debug session.
CPU DLL Driver DLL Parameter	Configures the internal µVision2 debug DLLs. The settings are taken from the device database. Please do not modify the DLL or DLL parameters.

Serial Window

µVision2 provides two Serial Windows for serial input and output. Serial data output from the simulated CPU is displayed in this window. Characters you type in the **Serial Window** are input to the simulated CPU.



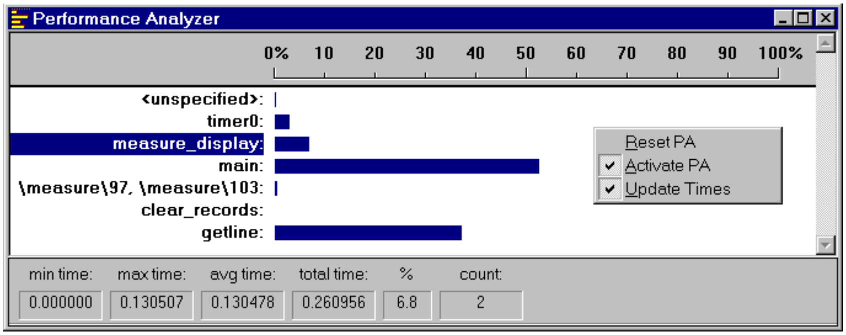
```
+*********** REMOTE MEASUREMENT RECORDER using C166 ***********+
| This program is a simple Measurement Recorder. It is based   |
| on the 80C166 CPU and records the state of Port 2 and the    |
| voltage on the four analog inputs AN0 through AN3.           |
+ command -+ syntax -----+ function ---------------------------+
| Read     | R [n]       | read <n> recorded measurements      |
| Display  | D           |                  surement values    |
| Time     | T hh:mm:ss  |                                     |
| Interval | I mm:ss.ttt |                                     |
| Clear    | C           |                  ecords             |
| Quit     | Q           | quit measurement recording          |
| Start    | S           | start measurement recording         |
+----------+-------------+-------------------------------------+

Command:
```

This lets you simulate the CPU's UART without the need for external hardware. The serial output may be also assigned to a PC COM port using the ASSIGN command in the **Output Window – Command** page.

Performance Analyzer

The µVision2 Performance Analyzer displays the execution time recorded for functions and address ranges you specify.



min time: 0.000000
max time: 0.130507
avg time: 0.130478
total time: 0.260956
%: 6.8
count: 2

The **<unspecified>** address range is automatically generated. It shows the amount of time spent executing code that is not included in the specified functions or address ranges.

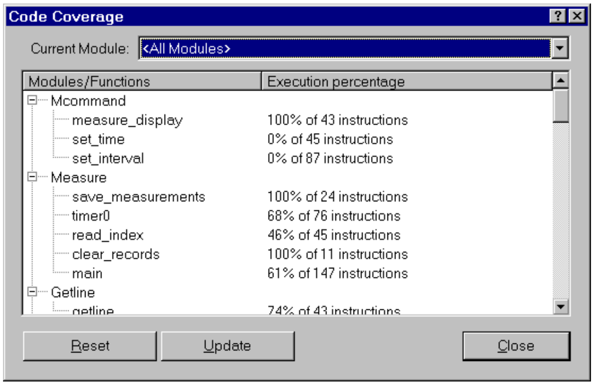
Results display as bar graphs. Information such as invocation count, minimum time, maximum time, and average time is displayed for the selected function or address range. Each of these statistics is described in the following table.

Label	Description
min time	The minimum time spent in the selected address range or function.
max time	The maximum time spent in the selected address range or function.
avg time	The average time spent in the selected address range or function.
total time	The total time spent in the selected address range or function.
%	The percent of the total time spent in the selected address range or function.
count	The total number of times the selected address range or function was executed.

To setup the Performance Analyzer use the menu command **Debug – Performance Analyzer**. You may enter the **PA** command in the command window to setup ranges or print results.

Code Coverage

The µVision2 debugger provides a code coverage function that marks the code that has been executed. In the debug window, lines of code that have been executed are market green in the left column. You can use this feature when you test your embedded application to determine the sections of code that have not yet been exercised.



Modules/Functions	Execution percentage
Mcommand	
measure_display	100% of 43 instructions
set_time	0% of 45 instructions
set_interval	0% of 87 instructions
Measure	
save_measurements	100% of 24 instructions
timer0	68% of 76 instructions
read_index	46% of 45 instructions
clear_records	100% of 11 instructions
main	61% of 147 instructions
Getline	
getline	74% of 43 instructions

The Code Coverage dialog provides information and statistics. You can output this information in the **Output Window – Command** page using the COVERAGE command.

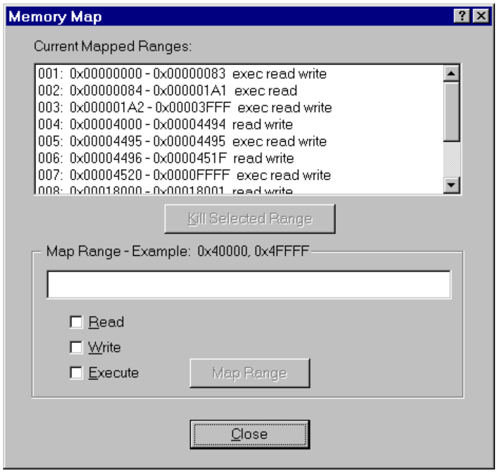
Memory Map

The Memory Map dialog box lets you specify the memory areas your target program uses for data storage and program execution. You may also configure the target program's memory map using the **MAP** command.

When you load a target application, µVision2 automatically maps all address ranges of your application. Typically it is not required to map additional address ranges. You need to map only memory areas that are accessed without explicit variable declarations, i.e. memory mapped I/O space.

The dialog opens via the menu **Debug – Memory Map**…

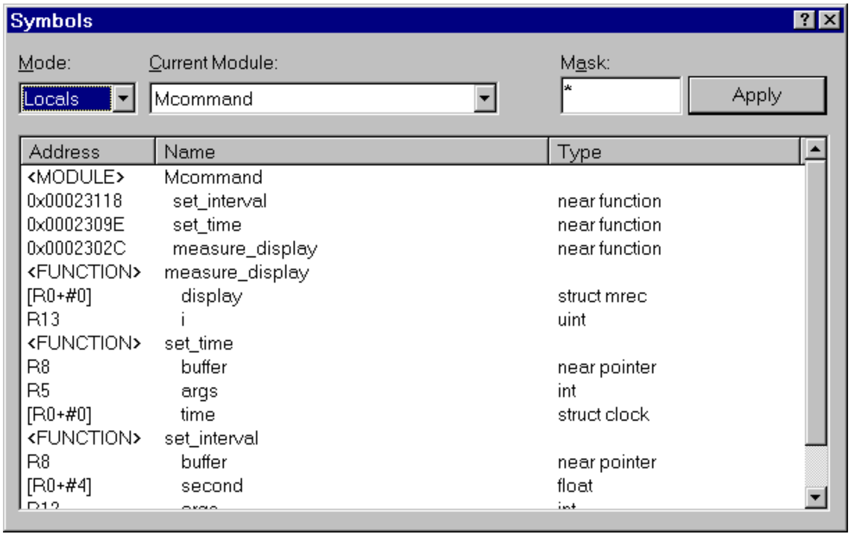
As your target program runs, µVision2 uses the memory map to verify that your program does not access invalid memory areas. For each memory range, you may specify the access method: **Read**, **Write**, **Execute**, or a combination.



```
001:  0x00000000 - 0x00000083  exec read write
002:  0x00000084 - 0x000001A1  exec read
003:  0x000001A2 - 0x00003FFF  exec read write
004:  0x00004000 - 0x00004494  read write
005:  0x00004495 - 0x00004495  exec read write
006:  0x00004496 - 0x0000451F  read write
007:  0x00004520 - 0x0000FFFF  exec read write
008:  0x00018000 - 0x00018001  read write
```

View – Symbols Window

The Symbols Window displays public symbols, local symbols or line number information defined in the currently loaded application program. CPU-specific SFR symbols are also displayed.



Mode: Locals

Current Module: Mcommand

Mask: *

Address	Name	Type
<MODULE>	Mcommand	
0x00023118	set_interval	near function
0x0002309E	set_time	near function
0x0002302C	measure_display	near function
<FUNCTION>	measure_display	
[R0+#0]	display	struct mrec
R13	i	uint
<FUNCTION>	set_time	
R8	buffer	near pointer
R5	args	int
[R0+#0]	time	struct clock
<FUNCTION>	set_interval	
R8	buffer	near pointer
[R0+#4]	second	float
R12	args	int

You may select the symbol type and filter the information with the options in the Symbol Window:

Options	Description
Mode	select PUBLIC, LOCALS or LINE. Public symbols have application-wide scope. The scope of local symbols is limited to a module or function. Lines refer to the line number information of the source text.
Current Module	select the source module where information should be displayed.
Mask	specify a mask that is used to match symbol names. The mask may consist of alphanumeric characters plus mask characters: **#** matches a digit (0 – 9) **$** matches any character ***** matches zero or more characters.
Apply	applies the mask and displays the update symbol list.

The following table provides a few examples of masks for symbol name.

Mask	Matches symbol names …
*****	Matches any symbol. This is the default mask in the Symbol Browser.
#	… that contain one digit in any position.
_a$#*	… with an underline, followed by the letter **a**, followed by any character, followed by a digit, ending with zero or more characters. For example, **_ab1** or **_a10value**.

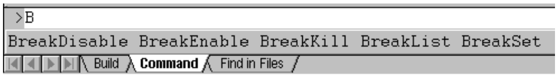
Mask	Matches symbol names …
_*ABC	… with an underline, followed by zero or more characters, followed by **ABC**.

Debug Commands

You may interact with the µVision2 debugger by entering commands in the **Output Window – Command** page. In the following tables all available µVision2 debug commands are listed in categories. Use the underlined characters in the command names to enter commands. For example, the **WATCHSET** command must be entered as **WS**.

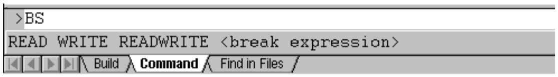
During command entry, the syntax generator displays possible commands, options and parameters. As you enter commands µVision2 reduces the list of likely commands to coincide with the characters you type.

If you type **B**, the syntax generator reduces the commands listed.



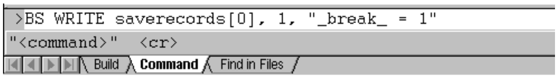
```
>B
BreakDisable BreakEnable BreakKill BreakList BreakSet
```

Available command options are listed if the command is clear.



```
>BS
READ WRITE READWRITE <break expression>
```

The syntax generator leads you through the command entry and helps you to avoid errors.



```
>BS WRITE saverecords[0], 1, "_break_ = 1"
"<command>"  <cr>
```

Memory Commands

The following memory commands let you display and alter memory contents.

Command	Description
ASM	Assembles in-line code.
DEFINE	Defines typed symbols that you may use with µVision2 debug functions.
DISPLAY	Display the contents of memory.
ENTER	Enters values into a specified memory area.
EVALUATE	Evaluates an expression and outputs the results.
MAP	Specifies access parameters for memory areas.
UNASSEMBLE	Disassembles program memory.
WATCHSET	Adds a watch variable to the Watch window.

Program Execution Commands

Program commands let you run code and step through your program one instruction at a time.

Command	Description
Esc	Stops program execution.
GO	Starts program execution.
PSTEP	Steps over instructions but does not step into procedures or functions.
OSTEP	Steps out of the current function.
TSTEP	Steps over instructions and into functions.

Breakpoint Commands

µVision2 provides breakpoints you may use to conditionally halt the execution of your target program. Breakpoints can be set on read operations, write operations and execution operations.

Command	Description
BREAKDISABLE	Disables one or more breakpoints.
BREAKENABLE	Enables one or more breakpoints.
BREAKKILL	Removes one or more breakpoints from the breakpoint list.
BREAKLIST	Lists the current breakpoints.
BREAKSET	Adds a breakpoint expression to the list of breakpoints.

General Commands

The following general commands do not belong in any other particular command group. They are included to make debugging easier and more convenient.

Command	Description
ASSIGN	Assigns input and output sources for the Serial window.
COVERAGE	List code coverage statistics.
DEFINE BUTTON	Creates a Toolbox button.
DIR	Generates a directory of symbol names.
EXIT	Exits the µVision2 debug mode.
INCLUDE	Reads and executes the commands in a command file.
KILL	Deletes µVision2 debug functions and Toolbox buttons.
LOAD	Loads CPU drivers, object modules, and HEX files.
LOG	Creates log files, queries log status, and closes log files for the Debug window.

Command	Description
MODE	Sets the baud rate, parity, and number of stop bits for PC COM ports.
PerformanceAnalyze	Setup the performance analyzer or list PA information.
RESET	Resets CPU, memory map assignments, Performance Analyzer or predefined variables.
SAVE	Saves a memory range in an Intel HEX386 file.
SCOPE	Displays address assignments of modules and functions of a target program.
SET	Sets the string value for predefined variable.
SETMODULE	Assigns a source file to a module.
SIGNAL	Displays signal function status and removes active signal functions.
SLOG	Creates log files, queries log status, and closes log files for the Serial window.

You can interactively display and change variables, registers, and memory locations from the command window. For example, you can type the following text commands at the command prompt:

Text	Effect
MDH	Display the **MDH** register.
R7 = 12	Assign the value 12 to register **R7**.
time.hour	Displays the member **hour** of the **time** structure.
time.hour++	Increments the member **hour** of the **time** structure.
index = 0	Assigns the value 0 to index.

Expressions

Many debug commands accept numeric expressions as parameters. A numeric expression is a number or a complex expressions that contains numbers, debug objects, or operands. An expression may consist of any of the following components.

Component	Description
Bit Addresses	Bit addresses reference bit-addressable data memory.
Constants	Constants are fixed numeric values or character strings.
Line Numbers	Line numbers reference code addresses of executable programs. When you compile or assemble a program, the compiler and assembler include line number information in the generated object module.
Operators	Operators include +, -, *, and /. Operators may be used to combine subexpressions into a single expression. You may use all operators that are available in the C programming language.
Program Variables (Symbols)	Program variables are those variables in your target program. They are often called symbols or symbolic names.

Component	Description
System Variables	System variables alter or affect the way µVision2 operates.
Type Specifications	Type specifications let you specify the data type of an expression or subexpression.

Constants

The µVision2 accepts decimal constants, HEX constants, octal constants, binary constants, floating-point constants, character constants, and string constants.

Binary, Decimal, HEX, and Octal Constants

By default, numeric constants are decimal or base ten numbers. When you enter `10`, this is the number ten and not the HEX value `10h`. The following table shows the prefixes and suffixes that are required to enter constants in base 2 (binary), base 8 (octal), base 10 (decimal), and base 16 (HEX).

Base	Prefix	Suffix	Example
Binary:	None	**Y** or **y**	**11111111Y**
Decimal:	None	**T** or none	**1234T** or **1234**
Hexadecimal:	**0x** or **0X**	**H** or **h**	**1234H** or **0x1234**
Octal:	None	**Q**, **q**, **O**, or **o**	**777q** or **777Q** or **777o**

Following are a few points to note about numeric constants.

- Numbers may be grouped with the dollar sign character (“$”) to make them easier to read. For example, `1111$1111y` is the same as `11111111y`.
- HEX constants must begin prefixed with a leading zero when the first digit in the constant is A-F.
- By default, numeric constants are 16-bit values. They may be followed with an `L` to make them long, 32-bit values. For example: `0x1234L`, `1234L`, `1255HL`.
- When a number is entered that is larger than the range of a 16-bit integer , the number is promoted automatically to a 32-bit integer.

Floating-Point Constants

Floating-point constants are entered in one of the following formats.

number **.** *number*

number **e**⟦+|-⟧ *number*

number **.** *number* ⟦**e**⟦+|-⟧ *number*⟧

For example, `4.12`, `0.1e3`, and `12.12e-5`. In contrast with the C programming language, floating-point numbers must have a digit before the decimal point. For example, `.12` is not allowed. It must be entered as `0.12`.

Character Constants

The rules of the C programming language for character constants apply to the µVision2 debugger. For example, the following are all valid character constants.

```
'a', '1', '\n', '\v', '\x0FE', '\015'
```

Also escape sequences are supported as listed in the following table:

Sequence	Description
\	Backslash character ("\").
\"	Double quote.
\'	Single quote.
\a	Alert, bell.
\b	Backspace.
\f	Form feed.

Sequence	Description
\n	Newline.
\r	Carriage return.
\t	Tab.
\0*nn*	Octal constant.
\X*nnn*	HEX constant.

String Constants

The rules of the C programming language for string constants also apply to µVision2. For example:

```
"string\x007\n"                                       "value of %s = %04XH\n"
```

Nested strings may be required in some cases. For example, double quotes for a nested string must be escaped. For example:

```
"printf (\"hello world!\n\")"
```

In contrast with the C programming language, successive strings are not concatenated into a single string. For example, **`"string1+" "string2"`** is not combined into a single string.

System Variables

System variables allow access to specific functions and may be used anywhere a program variable or other expression is used. The following table lists the available system variables, the data types, and their uses.

Variable	Type	Description
$	**unsigned long**	represents the program counter. You may use $ to display and change the program counter. For example, $ = 0x4000 sets the program counter to address 0x4000.
break	**unsigned int**	lets you stop executing the target program. When you set **_break_** to a non-zero value, µVision2 halts target program execution. You may use this variable in user and signal functions to halt program execution. Refer to "Chapter 6. µVision2 Debug Functions" on page 119 for more information.
traps	**unsigned int**	when you set **_traps_** to a non-zero value, µVision2 display messages for the 166 hardware traps: Undefined Opcode, Protected Instruction Fault, Illegal Word Operand Access, Illegal Instruction Access, Stack Underflow and Stack Overflow.
states	**unsigned long**	current value of the CPU instruction state counter; starts counting from 0 when your target program begins execution and increases for each instruction that is executed. ***NOTE**: **states** is a read-only variable.*
itrace	**unsigned int**	indicates whether or not trace recording is performed during target program execution. When **itrace** is 0, no trace recording is performed. When **itrace** has a non-zero value, trace information is recorded. Refer to page 82 for more information.
radix	**unsigned int**	determines the base used for numeric values displayed. **radix** may be 10 or 16. The default setting is 16 for HEX output.

On-chip Peripheral Symbols

µVision2 automatically defines a number of symbols depending on the CPU you have selected for your project. There are two types of symbols that are defined: special function registers (SFRs) and CPU pin registers (VTREGs).

Special Function Registers (SFRs)

µVision2 supports all special function registers of the microcontroller you have selected. Special function registers have an associated address and may be used in expressions.

CPU Pin Registers (VTREGs)

CPU pin registers, or VTREGs, let you use the CPU's simulated pins for input and output. VTREGs are not public symbols nor do they reside in a memory space of the CPU. They may be used in expressions, but their values and utilization are CPU dependent. VTREGs provide a way to specify signals coming into the CPU from a simulated piece of hardware. You can list these symbols with the **DIR VTREG** command.

The following table describes the VTREG symbols. The VTREG symbols that are actually available depend on the selected CPU.

VTREG	Description
AIN*x*	An analog input pin on the chip. Your target program may read values you write to **AIN*x*** VTREGs.
PORT*x*	A group of I/O pins for a port on the chip. For example, **PORT2** refers to all 8 or 16 pins of P2. These registers allow you to simulate port I/O.
S*x*IN	The input buffer of serial interface ***x***. You may write 8-bit or 9-bit values to **S*x*IN**. These are read by your target program. You may read **S*x*IN** to determine when the input buffer is ready for another character. The value 0xFFFF signals that the previous value is completely processed and a new value may be written.
S*x*OUT	The output buffer of serial interface ***x***. µVision2 copies 8-bit or 9-bit values (as programmed) to the **S*x*OUT** VTREG.
S*x*TIME	Defines the baudrate timing of the serial interface ***x***. When **S*x*TIME** is 1, µVision2 simulates the timing of the serial interface using the programmed baudrate. When **S*x*TIME** is 0 (the default value), the programmed baudrate timing is ignored and serial transmission time is instantaneous.
XTAL	The XTAL frequency of the simulated CPU as defined in the **Options – Target** dialog.

NOTE
You may use the VTREGs to simulate external input and output including interfacing to internal peripherals like interrupts and timers. For example, if you toggle bit 2 of PORT3 (on the 8051 drivers), the CPU driver simulates external interrupt 0.

For the C517 CPU the following VTREG symbols for the on-chip peripheral registers are available:

CPU-pin Symbol	Description
AIN0	Analog input line AIN0 (floating-point value)
AIN1	Analog input line AIN1 (floating-point value)
AIN2	Analog input line AIN2 (floating-point value)
AIN3	Analog input line AIN3 (floating-point value)

CPU-pin Symbol	Description
AIN4	Analog input line AIN4 (floating-point value)
AIN5	Analog input line AIN5 (floating-point value)
AIN6	Analog input line AIN6 (floating-point value)
AIN7	Analog input line AIN7 (floating-point value)
AIN8	Analog input line AIN8 (floating-point value)
AIN9	Analog input line AIN9 (floating-point value)
AIN10	Analog input line AIN10 (floating-point value)
AIN11	Analog input line AIN11 (floating-point value)
PORT0	Digital I/O lines of PORT 0 (8-bit)
PORT1	Digital I/O lines of PORT 1 (8-bit)
PORT2	Digital I/O lines of PORT 2 (8-bit)
PORT3	Digital I/O lines of PORT 3 (8-bit)
PORT4	Digital I/O lines of PORT 4 (8-bit)
PORT5	Digital I/O lines of PORT 5 (8-bit)
PORT6	Digital I/O lines of PORT 6 (8-bit)
PORT7	Digital I/O lines of PORT 7 (8-bit)
PORT8	Digital I/O lines of PORT 8 (8-bit)
S0IN	Serial input for SERIAL CHANNEL 0 (9-bit)
S0OUT	Serial output for SERIAL CHANNEL 0 (9-bit)
S1IN	Serial input for SERIAL CHANNEL 1 (9-bit)
S1OUT	Serial output for SERIAL CHANNEL 1 (9-bit)
STIME	Serial timing enable
VAGND	Analog reference voltage VAGND (floating-point value)
VAREF	Analog reference voltage VAREF (floating-point value)
XTAL	Oscillator frequency

The following examples show how VTREGs may be used to aid in simulating your target program. In most cases, you use VTREGs in signal functions to simulate some part of your target hardware.

I/O Ports

µVision2 defines a VTREG for each I/O port: i.e. **PORT2**. Do not confuse these VTREGs with the SFRs for each port (i.e. **P2**). The SFRs can be accessed inside the CPU memory space. The VTREGs are the signals present on the pins.

With µVision2, it is easy to simulate input from external hardware. If you have a pulse train coming into a port pin, you can use a signal function to simulate the signal. For example, the following signal function inputs a square wave on P2.1 with a frequency of 1000Hz.

```
signal void one_thou_hz (void) {
  while (1) {                              /* repeat forever */
    PORT2 |= 1;                            /* set P1.2 */
    twatch ((CLOCK / 2) / 2000);           /* delay for .0005 secs */
    PORT2 &= ~1;                           /* clear P1.2 */
    twatch ((CLOCK / 2) / 2000);           /* delay for .0005 secs */
  }                                        /* repeat */
}
```

The following command starts this signal function:

```
one_thou_hz ()
```

Refer to “Chapter 6. µVision2 Debug Functions” on page 119 for more information about user and signal functions.

Simulating external hardware that responds to output from a port pin is only slightly more difficult. Two steps are required. First, write a µVision2 user or signal function to perform the desired operations. Second, create a breakpoint that invokes the user function.

Suppose you use an output pin (P2.0) to enable or disable an LED. The following signal function uses the **PORT2** VTREG to check the output from the CPU and display a message in the Command window.

```
signal void check_p20 (void) {
  if (PORT2 & 1)) {                     /* Test P2.0 */
    printf ("LED is ON\n"); }          /* 1? LED is ON */
  else {                                /* 0? LED is OFF */
    printf ("LED is OFF\n"): }
}
```

Now, you must add a breakpoint for writes to port 1. The following command line adds a breakpoint for all writes to PORT2.

```
BS WRITE PORT2, 1, "check_p20 ()"
```

Now, whenever your target program writes to PORT2, the check_P20 function prints the current status of the LED. Refer to page 83 for more information about setting breakpoints.

Serial Ports

The on-chip serial port is controlled with: **S0TIME**, **S0IN**, and **S0OUT**. **S0IN** and **S0OUT** represent the serial input and output streams on the CPU. **S0TIME** lets you specify whether the serial port timing instantaneous (**STIME** = 0) or the serial port timing is relative to the specified baudrate (**S*x*TIME** = 1). When **S0TIME** is 1, serial data displayed in the Serial window is output at the specified baudrate. When **S0TIME** is 0, serial data is displayed in the Serial window much more quickly.

Simulating serial input is just as easy as simulating digital input. Suppose you have an external serial device that inputs specific data periodically (every second). You can create a signal function that feeds the data into the CPU’s serial port.

```
signal void serial_input (void) {
  while (1) {                            /* repeat forever */
    twatch (CLOCK);                      /* Delay for 1 second */

    S0IN = 'A';                          /* Send first character */
    twatch (CLOCK / 900);                /* Delay for 1 character time */
                                         /* 900 is good for 9600 baud */
    S0IN = 'B';                          /* Send next character */
    twatch (CLOCK / 900);
    S0IN = 'C';                          /* Send final character */
  }                                      /* repeat */
}
```

When this signal function runs, it delays for 1 second, inputs ‘A’, ‘B’, and ‘C’ into the serial input line and repeats.

Serial output is simulated in a similar fashion using a user or signal function and a write access breakpoint as described above.

Program Variables (Symbols)

µVision2 lets you access variables, or symbols, in your target program by simply typing their name. Variable names, or symbol names, represent numeric values and addresses. Symbols make the debugging process easier by allowing you to use the same names in the debugger as you use in your program.

When you load a target program module and the symbol information is loaded into the debugger. The symbols include local variables (declared within functions), the function names, and the line number information. You must enable **Options for Target – Output – Debug Information**. Without debug information, µVision2 cannot perform source-level and symbolic debugging.

Module Names

A module name is the name of an object module that makes up all or part of a target program. Source-level debugging information as well as symbolic information is stored in each module.

The module name is derived from the name of the source file. If the target program consists of a source file named **MCOMMAND.C** and the C compiler generates an object file called **MCOMMAND.OBJ**, the module name is **MCOMMAND**.

Symbol Naming Conventions

The following conventions apply to symbols.

- The case of symbols is ignored: **SYMBOL** is equivalent to **Symbol**.
- The first character of a symbol name must be: ‘A’-’Z’, ‘a’-’z’, ‘_’, or ‘?’.
- Subsequent characters may be: ‘A’-’Z’, ‘a’-’z’, ‘0’-’9’, ‘_’, or ‘?’.

NOTE
When using the ternary operator (“?:”) in µVision2 with a symbol that begins with a question mark (“?”), you must insert a space between the ternary operator and the symbol name. For example, ***R5 = R6 ? ?symbol : R7****.*

Fully Qualified Symbols

Symbols may be entered using a fully qualified name that includes the name of the module and name of the function in which the symbol is defined. A fully qualified symbol name is composed of the following components:

- **Module Name** identifies the module where a symbol is defined.
- **Line Number** identifies the address of the code generated for a particular line in the module.
- **Function Name** identifies the function in a module where a local symbol is defined.
- **Symbol Name** identifies the name of the symbol.

This components may combined as shown in the following table:

Symbol Components	Full Qualified Symbol Name addresses …
\ModuleName\LineNumber	… line number ***LineNumber*** in ***ModuleName***.
\ModuleName\FunctionName	… ***FunctionName*** function in ***ModuleName***.
\ModuleName\SymbolName	… global symbol ***SymbolName*** in ***ModuleName***.
\ModuleName\FunctionName\SymbolName	… local symbol ***SymbolName*** in the ***FunctionName*** function in ***ModuleName***.

Examples of fully qualified symbol names:

Full Qualified Symbol Name	Identifies …
\MEASURE\clear_records\idx	… local symbol **idx** in the **clear_records** function in the **MEASURE** module.
\MEASURE\MAIN\cmdbuf	… **cmdbuf** local symbol in the **MAIN** function in the **MEASURE** module.
\MEASURE\sindx	… **sindex** symbol in the **MEASURE** module.
\MEASURE\225	… line number 225 in the **MEASURE** module.
\MCOMMAND\82	… line number 82 in the **MCOMMAND** module.
\MEASURE\TIMER0	… the **TIMER0** symbol in the **MEASURE** module. This symbol may be a function or a global variable.

Non-Qualified Symbols

Symbols may be entered using the only name of the variable or function they reference. These symbols are not fully qualified and searched in a number of tables until a matching symbol name is found. This search works as follows:

1. **Register Symbols** of the CPU: R0 – R15, RL0 – RH7, DPP0 – DPP3.

2. **Local Variables in the Current Function** in the target program. The current function is determined by the value of the program counter.
3. **Static Variables in the Current Module**. As with the current function, the current module is determined by the value of the program counter. Symbols in the current module represent variables that were declared in the module but outside a function.
4. **Global or Public Symbols** of your target program. SFR symbols defined by µVision2 are considered to be public symbols and are also searched.
5. **Symbols Created with the µVision2 DEFINE Command**. These symbols are used for debugging and are not a part of the target program.
6. **System Variables** that monitor and change debugger characteristics. They are not a part of the target program. Refer to “System Variables” on page 101 for more information.
7. **CPU Driver Symbols (VTREGs)** defined by the CPU driver. Refer to “CPU Pin Registers (VTREGs)” on page 102 for a description of VTREG symbols.

NOTES

The search order for symbols changes when creating user or signal functions. µVision2 first searches the table of symbols defined in the user or signal function. Then, the above list is searched. Refer to “Chapter 6. µVision2 Debug Functions” on page 119 for more information about user and signal functions.

A literal symbol that is preceded with a back quote character (`) modifies the search order: ***CPU driver symbols (VTREGs)*** *are searched instead of* ***CPU register symbols****.*

Literal Symbols

With the back quote character (`) you get a literal symbol name. Literal symbols must be used to access:

- a program variable or symbol which is identical with a predefined **Reserved Word**. **Reserved Words** are µVision2 debug commands & options, data type names, CPU register names and assembler mnemonics.
- a CPU driver symbol (VTREG) that is identical to program variable name.

If a literal symbol name is given, µVision2 changes the search order for non-qualified symbols that is described above. For a literal symbol **CPU Driver Symbols (VTREGs)** are searched instead of **CPU Register Symbols**.

Examples for using Literal Symbols

If you define a variable named ***R5*** in your program and you attempt to access it, you will actually access the **R5** CPU register. To access the ***R5*** variable, you must prefix the variable name with the back quote character.

Accessing the R5 Register	**Accessing the R5 Variable**
`>R5 = 121`	``>`R5 = 212``

If your program contains a function named ***clock*** and you attempt to **clock** VTREG, you will get the address of the ***clock*** function. To access the **clock** VTREG, you must prefix the variable name with the back quote character.

Accessing the *clock* function	**Accessing the clock VTREG**

`>clock`
`0x00000DB2`
	``>`clock``
	`20000000`

Line Numbers

Line numbers enable source-level debugging and are produced by the compiler or assembler. The line number specifies the physical address in the source module of the associated program code. Since a line number represents a code address, µVision2 lets you use in an expression. The syntax for a line number is shown in the following table.

Line Number Symbol	Code Address …
\LineNumber	… for line number ***LineNumber*** in the current module.
\ModuleName\LineNumber	… for line number ***LineNumber*** in ***ModuleName***.

Example

```
\measure\108                 /* Line 108 in module "MEASURE" */
\143                         /* Line 143 in the current module */
```

Bit Addresses

Bit addresses represent bits in the memory. This includes bits in special function registers. The syntax for a bit address is ***expression . bit_position***

Examples

```
R6.2                          /* Bit 2 of register R6 */
0xFD00.15                     /* Value of the 166 bit space */
```

Type Specifications

µVision2 automatically performs implicit type casting in an expression. You may explicitly cast expressions to specific data types. Type casting follows the conventions used in the C programming language. Example:

```
(unsigned int) 31.2    /* gives unsigned int 31 from the float value */
```

Operators

µVision2 supports all operators of the C programming language. The operators have the same meaning as their C equivalents.

Differences Between µVision2 and C

There are a number of differences between expressions in µVision2 and expressions in the C programming language:

- µVision2 does not differentiate between uppercase and lowercase characters for symbolic names and command names.
- µVision2 does not support converting an expression to a typed pointer like **char *** or **int ***. Pointer types are obtained from the symbol information in the target program. They cannot be created.
- Function calls entered in the µVision2 Output Window – Command page refer to debug functions. You cannot invoke functions in your target from the command line. Refer to “Chapter 6. µVision2 Debug Functions” on page 119 for more information.
- µVision2 does not support structure assignments.

Expression Examples

The following expressions were entered in the Command page of the Output Window. All applicable output is included with each example. The MEASURE example program were used for all examples.

Constant

```
>0x1234                                               /* Simple constant */
0x1234                                                         /* Output */
>EVAL 0x1234
4660T 11064Q 1234H '...4'              /* Output in several number bases */
```

Register

```
>R1                                  /* Interrogate value of register R1 */
0x000A                        /* Address from ACC = 0xE0, mem type = D:  */
>R1 = --R7                         /* Set R1 and R7 equal to value R7-1  */
```

Function Symbol

```
>main                            /* Get address of main() from MEASURE.C */
0x00233DA                               /* Reply, main starts at 0x233DA */

>&main                                                 /* Same as before */
0x00233DA

>d main                                       /* Display: address = main */
0x0233DA: 76 E2 00 04 76 E3 00 04 - 66 E3 FF F7 E6 B6 80 00 v...v...f......
0x0233EA: E6 B7 00 00 E6 5A 40 00 - E6 D8 11 80 E6 2A 3C F6 .....Z@......*<
0x0233FA: E6 28 3C F6 E6 CE 44 00 - BF 88 E6 A8 40 00 BB D8 .(<...D.....@..
0x02340A: E6 F8 7A 40 CA 00 CE 39 - E6 F8 18 44 CA 00 CE 39 ..z@...9...D...
```

Address Utilization Examples

```
>&\measure\main\cmdbuf[0] + 10                    /* Address calculation */
0x23026

>_RBYTE (0x233DA)                 /* Read byte from code address 0x233DA */
0x76                                                            /* Reply */
```

Symbol Output Examples

```
>dir \measure\main       /* Output symbols from main() in module MEASURE */
    R14  idx . . . uint                                        /* Output */
    R13  i . . . uint
    0x0002301C   cmdbuf . . . array[15] of char
```

Program Counter Examples

```
>$ = main                              /* Set program counter to main()  */
>dir                            /* points to local mem sym.  from main() */
R14  idx . . . uint                                            /* Output */
    R13  i . . . uint
    0x0002301C   cmdbuf . . . array[15] of char
```

Program Variable Examples

```
>cmdbuf                               /* Interrogate address from cmdbuf */
0x0002301C             /* Output of address due to aggregate type (Array)*/
>cmdbuf[0]                     /* Output contents of first array element */
0x00
>i                                               /* Output contents from i */
0x00
>idx                                           /* Output contents from idx */
```

```
0x0000
>idx = DPP2            /* Set contents from index equal to register DPP2 */
>idx                                         /* Output contents from idx */
0x0008
```

Line Number Examples

```
>\163                                 /* Address of the line number #104 */
0x000230DA                                                      /* Reply */
>\MCOMMAND\91                      /* A line number of module "MCOMMAND" */
0x000231F6
```

Operator Examples

```
>--R5                           /* Auto-decrement also for CPU registers */
0xFE
>mdisplay                                /* Output a PUBLIC bit variable */
0
>mdisplay = 1                                                  /* Change */
>mdisplay                                                /* Check result */
1
```

Structure Examples

```
>save_record[0]                                   /* Address of a record */
0x002100A
>save_record[0].time.hour = DPP3     /* Change struct element of records */

>save_record[0].time.hour                                /* Interrogation */
0x03
```

5

µVision2 Debug Function Invocation Examples

```
>printf ("uVision2 is coming!\n")     /* String constant within printf() */
uVision2 is coming!                                            /* Output */
>_WBYTE(0x20000, _RBYTE(0x20001))            /* Read & Write Memory Byte */
>                                   /* example useful in debug functions */
>interval.min = getint ("enter integer:  ");
```

Fully Qualified Symbol Examples

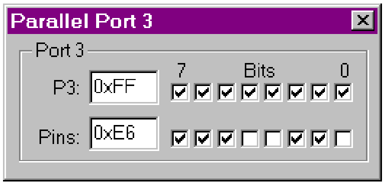
```
>--\measure\main\idx          /* Auto INC/DEC valid for qualified symbol */
0xFFFF
```

Tips and Tricks

The following section discusses advanced techniques that you may use with the µVision2 debugger. You will not need the following features very often, but readers of this section get a better feeling for the µVision2 debugger capabilities.

Simulate I/O Ports

µVision2 provides dialogs that show the status of all I/O ports. The I/O Pins are represented with VTREGs. You may use this VTREGs also together with signal functions or breakpoints as shown in the following example program.



P3: 0xFF ☑ ☑ ☑ ☑ ☑ ☑ ☑ ☑

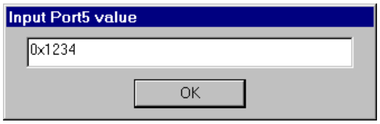
Pins: 0xE6 ☑ ☑ ☑ □ □ ☑ ☑ □

```
// in your C user program
  p1value = P1;    // read Port 1 input
  P3 = p1value;    // write to Port 3
```

Breakpoints that you define in the µVision2 simulator:

```
bs write PORT3, 1, "printf (\"Port3 value=%X\\n\", PORT3)"
bs read  PORT1, 1, "PORT1 = getint (\"Input Port1 value\")"
```

When you execute your C program, µVision2 asks you for a Port1 input value. If a new output value is written to Port3, a message is printed in the **Output Window - Command** page. Refer also to “CPU Pin Registers (VTREGs)” on page 102.



0x1234

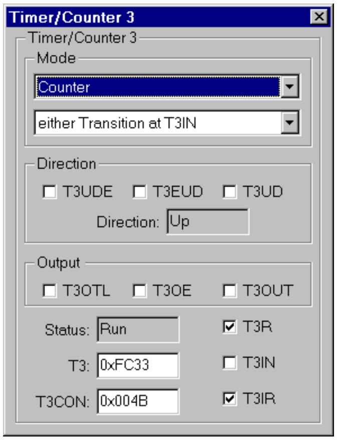
Simulate Interrupts and Clock Inputs

µVision2 simulates the behavior of the I/O inputs. If an I/O pin is configured as counter input the count value increments when the pin toggles. The following example shows how to simulate input for Counter 3:

```
// in your C user program
T3CON = 0x004B;   // set T3 Counter Mode
```

You may toggle the counter input P3.6 with the VTREG PORT3, i.e. with a signal function:

```
signal void ToggleT3Input (void)  {
  while (1)  {
    PORT3 = PORT3 ^ 0x40;     // toggle P3.6
    twatch (CLOCK / 100000); // with 100kHz
  }
}
```



Mode: Counter

either Transition at T3IN

□ T3UDE □ T3EUD □ T3UD

Direction: Up

□ T3OTL □ T3OE □ T3OUT

Status: Run ☑ T3R

T3: 0xFC33 □ T3IN

T3CON: 0x004B ☑ T3IR

View the Counter 3 status with the Peripheral dialog.

Also interrupt inputs are simulated: if a port pin is used as interrupt input, the interrupt request will be set if you toggle the associated I/O pin.

Simulate external I/O Devices

External I/O devices are typical memory mapped. You may simulate such I/O devices with the **Memory Window** provided in the µVision2 debugger. Since the C user program does not contain any variable declarations for such memory regions it is required that you map this memory with the MAP command:

```
MAP X:0x1000, X:0x1FFF READ WRITE            /* MAP memory for I/O area */
```

You may use breakpoints in combination with debug functions to simulate the logic behind the I/O device. Refer to “User Functions” on page 132 for more information. Example for a breakpoint definition:

```
BS WRITE 0x100000, 1, "IO_access ()"
```

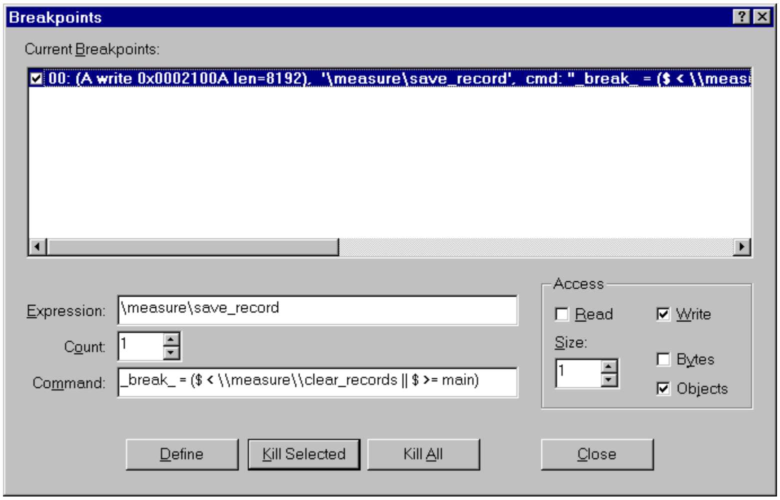
Assign Serial I/O to a PC COM Port

The ASSIGN command allows you to use a PC COM Port as input for an UART in the µVision2 simulator. If you enter the following commands, serial I/O is performed via the COM2: interface of your PC. The STIME variable allows you to ignore the timing of the simulated serial interface.

```
>MODE COM2 9600, 0, 8, 1   /*9600 bps, no parity, 8 data & 1 stop bit*/
>ASSIGN COM2 <S0IN >S0OUT  /*ASC0 output & input is done with COM2:*/
>S0TIME = 0                /*ignore timing of simulated ASC0 interface*/
```

Check Illegal Memory Accesses

Sometimes it is required to trap illegal memory accesses. The µVision2 access breakpoints might be used together with the system variable **_break_**. In the following example the program execution stops when the array ***save_record*** is accessed outside of the function ***clear_records***.



Breakpoints

Current Breakpoints:

☑ 00: (A write 0x0002100A len=8192), '\measure\save_record', cmd: "_break_ = ($ < \\meas

Expression: \measure\save_record

Count: 1

Command: _break_ = ($ < \\measure\\clear_records || $ >= main)

Access

□ Read ☑ Write

Size: 1

□ Bytes

☑ Objects

Command Input from File

Commands for the µVision2 debugger might be read from file with the INCLUDE command. Under **Options for Target - Debug** you may also specify an **Initialization File** with debug commands. Refer to page 90 for more information.

Preset I/O Ports or Memory Contents

Some applications require that I/O port values or memory contents are set to specific values before program simulation. In the debug **Initialization File** you may include the commands that are required to preset the simulator. Example:

```
PORT3 = 0                 /* set Port3 to zero                          */
LOAD MEMORY.HEX           /* load hex file contents to memory           */
                          /* use the SAVE command to save memory contents */
```

Write Debug Output to a File

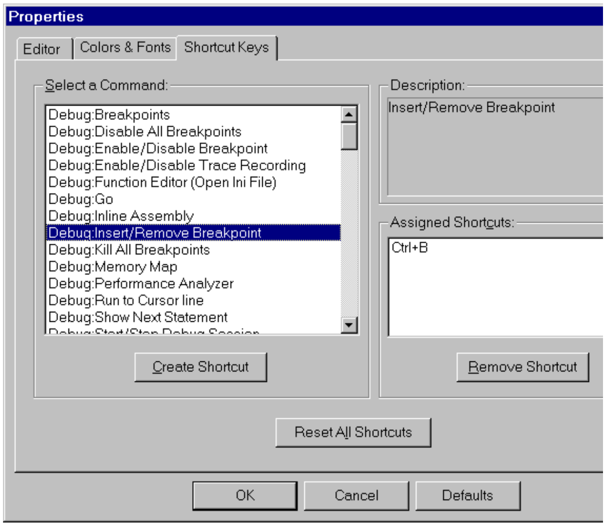
The commands **LOG** and **SLOG** can be used to write debug output files. You may run the µVision2 debugger in batch files and use a debug **Initialization File** that contains these commands to automate program test. Refer to “µVision 2 Command Line Invocation” on page 177 for additional information.

```
>LOG >>C:\TMP\DEBUGOUT.TXT      /* protocol Output Window - Command page*/
>SLOG >>C:\TMP\DEBUGOUT.TXT     /* protocol Serial Window output        */
>/* Output of the Command page and the Serial Window is written to file */
>LOG OFF                        /* stop Output Window protocol          */
>SLOG OFF                       /* stop Serial Window protocol          */
```

Use Keyboard Shortcuts

View – Options allows you to configure shortcut keys for all menu items. With this dialog you may personalize µVision2 to your needs. For example, you may add a shortcut key to insert/remove breakpoints in an editor window.

NOTE
The assignment of shortcut keys is saved in the file ***C:\KEIL\UV2\UV2.MAC****.*


Properties
Editor
Colors & Fonts
Shortcut Keys
Select a Command:
Debug:Breakpoints
Debug:Disable All Breakpoints
Debug:Enable/Disable Breakpoint
Debug:Enable/Disable Trace Recording
Debug:Function Editor (Open Ini File)
Debug:Go
Debug:Inline Assembly
Debug:Insert/Remove Breakpoint
Debug:Kill All Breakpoints
Debug:Memory Map
Debug:Performance Analyzer
Debug:Run to Cursor line
Debug:Show Next Statement
Debug:Start/Stop Debug Session
Create Shortcut
Remove Shortcut
Reset All Shortcuts
OK
Cancel
Defaults

Description: Insert/Remove Breakpoint

Assigned Shortcuts: Ctrl+B

Kernel Aware Debugging

µVision2 supports Kernel Awareness for operating systems with debug DLLs. Refer to “RTX Kernel Aware Debugging” on page 167 for details on testing programs that use the RTX-51 Tiny real-time operating system. RTX-51 Full applications are tested with similar features. µVision2 allows you to add own debug DLLs that display the status information for operating systems or other applications. We will provide an **Application Note** on www.keil.com that explains how to write user-specific debug DLLs for the µVision2 debugger.

Chapter 6. µVision2 Debug Functions

This chapter discusses a powerful aspect of the µVision2: debug functions. You may use functions to extend the capabilities of the µVision2 debugger. You may create functions that generate external interrupts, log memory contents to a file, update analog input values periodically, and input serial data to the on-chip serial port.

NOTE
*Do note confuse µVision2 debug functions with functions of your target program. µVision2 debug functions aids you in debugging of your application and are entered or with the **Function Editor** or on µVision2 command level.*

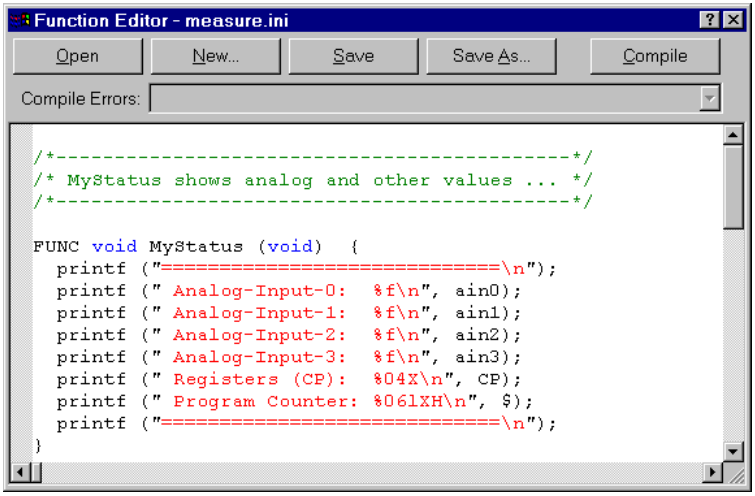
µVision2 debug functions utilize a subset of the C programming language. The basic capabilities and restrictions are as follows:

- Flow control statements **if**, **else**, **while**, **do**, **switch**, **case**, **break**, **continue**, and **goto** may be used in debug functions. All of these statements operate in µVision2 debug functions as they do in ANSI C.
- Local scalar variables are declared in debug functions in the same way they are declared in ANSI C. Arrays are not allowed in debug functions.

For a complete description of the "Differences Between Debug Functions and C" refer to page 138.

Creating Functions

µVision2 has a built-in debug function editor which opens with **Debug – Function Editor**. When you start the function editor, the editor asks for a file name or opens the file specified under **Options for Target – Debug – Initialization File**. The debug function editor works in the same way as the build-in µVision2 editor and allows you to enter and compile debug functions.



```
/*--------------------------------------------*/
/* MyStatus shows analog and other values ... */
/*--------------------------------------------*/

FUNC void MyStatus (void)  {
  printf ("=============================\n");
  printf (" Analog-Input-0:  %f\n", ain0);
  printf (" Analog-Input-1:  %f\n", ain1);
  printf (" Analog-Input-2:  %f\n", ain2);
  printf (" Analog-Input-3:  %f\n", ain3);
  printf (" Registers (CP):  %04X\n", CP);
  printf (" Program Counter: %06lXH\n", $);
  printf ("=============================\n");
}
```

Options	Description
Open	open an existing file with µVision2 debug functions or commands.
New	create a new file
Save	save the editor content to file.
Save As	specify a file for saving the debug functions.
Compile	send current editor content to the µVision2 command interpreter. This compiles all debug functions.
Compile Errors	shows a list of all errors. Choose an error, this locates the cursor to the erroneous line in the editor window.

Once you have created a file with µVision2 debug functions, you may use the **INCLUDE** command to read and process the contents of the text file. For example, if you type the following command in the command window, µVision2 reads and interprets the contents of **MYFUNCS.INI**.

```
>INCLUDE MYFUNCS.INI
```

MYFUNCS.INI may contain debug commands and function definitions. You may enter this file also under **Options for Target – Debug - Initialization File**. Every time you start the µVision2 debugger, the contents of **MYFUNCS.INI** will be processed.

Functions that are no longer needed may be deleted using the **KILL** command.

Invoking Functions

To invoke or run a debug function you must type the name of the function and any required parameters in the command window. For example, to run the **printf** built-in function to print “Hello World,” enter the following text in the command window:

```
>printf ("Hello World\n")
```

The µVision2 debugger responds by printing the text “Hello World” in the Command page of the Output Window.

Function Classes

µVision2 supports the following three classes of functions: Predefined Functions, User Functions, and Signal Functions.

- **Predefined Functions** perform useful tasks like waiting for a period of time or printing a message. Predefined functions cannot be removed or redefined.
- **User Functions** extend the capabilities of µVision2 and can process the same expressions allowed at the command level. You may use the predefined function **exec**, to execute debug commands from user and signal functions.
- **Signal Functions** simulate the behavior of a complex signal generator and lets you create various input signals to your target application. For example, signals can be applied on the input lines of the CPU under simulation. Signal functions run in the background during your target program’s execution. Signal functions are coupled via CPU states counter which has a resolution of instruction state. A maximum of 64 signal functions may be active simultaneously.

As functions are defined, they are entered into the internal table of user or signal functions. You may use the **DIR** command to list the predefined, user, and signal functions available.

DIR BFUNC displays the names of all built-in functions. **DIR UFUNC** displays the names of all user functions. **DIR SIGNAL** displays the names of all signal functions. **DIR FUNC** displays the names of all user, signal, and built-in functions.

Predefined Functions

µVision2 includes a number of predefined debug functions that are always available for use. They cannot be redefined or deleted. Predefined functions are provided to assist the user and signal functions you create.

The following table lists all predefined µVision2 debug functions.

Return	Name	Parameter	Description
void	**exec**	(“***command_string***”)	Execute Debug Command
double	**getdbl**	(“***prompt_string***”)	Ask the user for a double number
int	**getint**	(“***prompt_string***”)	Ask the user for a int number
long	**getlong**	(“***prompt_string***”)	Ask the user for a long number
void	**memset**	(***start_addr***, ***value, len***)	fill memory with constant value
void	**printf**	(“***string***”, ...)	works like the ANSI C printf function
int	**rand**	(**int** ***seed***)	return a random number in the range -32768 to +32767
void	**twatch**	(**ulong** ***states***)	Delay execution of signal function for specified number of CPU states
int	**_TaskRunning_**	(**ulong** *func_address*)	Checks if the specified task function is the current running task. Only available if a DLL for RTX Kernel Awareness is used.
uchar	**_RBYTE**	(***address***)	Read **char** on specified memory address
uint	**_RWORD**	(***address***)	Read **int** on specified memory address
ulong	**_RDWORD**	(***address***)	Read **long** on specified memory address
float	**_RFLOAT**	(***address***)	Read **float** on specified memory address
double	**_RDOUBLE**	(***address***)	Read **double** on specified memory address
void	**_WBYTE**	(***address,*** **uchar** ***val***)	Write **char** on specified memory address
void	**_WWORD**	(***address,*** **uint** ***val***)	Write **int** on specified memory address
void	**_WDWORD**	(***address,*** **ulong** ***val***)	Write **long** on specified memory address
void	**_WFLOAT**	(***address,*** **float** ***val***)	Write **float** on specified memory address
void	**_WDOUBLE**	(***address,*** **double** ***val***)	Write **double** on specified memory address

The predefined functions are described below.

void exec (“command_string”)

The **exec** function lets you invoke µVision2 debug commands from within your user and signal functions. The *command_string* may contain several commands separated by semicolons.

The *command_string* is passed to the command interpreter and must be a valid debug command.

Example

```
>exec ("DIR PUBLIC; EVAL R7")
>exec ("BS timer0")
>exec ("BK *")
```

double getdbl (“*prompt_string*”), int getint (“*prompt_string*”), long getlong (“*prompt_string*”)

This functions prompts you to enter a number and, upon entry, returns the value of the number entered. If no entry is made, the value 0 is returned.

Example

```
>age = getint ("Enter Your Age")
```

void memset (*start address*, *uchar value, ulong length*)

The **memset** function sets the memory specified with start address and length to the specified value.

Example

```
>MEMSET (0x20000, 'a', 0x1000)       /* Fill 0x20000 to 0x20FFF with "a" */
```

void printf (“format_string”, ...)

The **prinf** function works like the ANSI C library function. The first argument is a format string. Following arguments may be expressions or strings. The conventional ANSI C formatting specifications apply to **printf**.

Example

```
>printf ("random number = %04XH\n", rand(0))
random number = 1014H

>printf ("random number = %04XH\n", rand(0))
random number = 64D6H

>printf ("%s for %d\n", "uVision2", 166)
uVision2 for 166

>printf ("%lu\n", (ulong) -1)
4294967295
```

int rand (int *seed*)

The **rand** function returns a random number in the range -32768 to +32767. The random number generator is reinitialized each time a non-zero value is passed in the *seed* argument. You may use the **rand** function to delay for a random number of clock cycles or to generate random data to feed into a particular algorithm or input routine.

Example

```
>rand (0x1234)                /* Initialize random generator with 0x1234 */
0x3B98

>rand (0)                                          /* No initialization */
0x64BD
```

void twatch (long *states*)

The **twatch** function may be used in a signal function to delay continued execution for the specified number of CPU *states*. µVision2 updates the state counter while executing your target program.

Example

The following signal function toggles the INT0 input (P3.2) every second.

```
signal void int0_signal (void) {
  while (1) {
    PORT3 |=  0x04;       /* pull INT0(P3.2) high */
    PORT3 &= ~0x04;       /* pull INT0(P3.2) low and generate interrupt */
    PORT3 |=  0x04;       /* pull INT0(P3.2) high again */
    twatch (CLOCK);       /* wait for 1 second */
    }
  }
```

NOTE
*The **twatch** function may be called only from within a signal function. Calls outside a signal function are not allowed and result in an error message.*

int _TaskRunning_ (ulong *func_address*)

This function checks if the specified task function is the current running task. **_TaskRunning_** is only available if you select an **Operating System** under **Options for Target – Target**. µVision2 loads an additional DLL that kernel awareness for operating systems. Refer to “RTX Kernel Aware Debugging” on page 167 for more information.

The result of the debug function **_TaskRunning_** may be assigned to the **_break_** system variable to stop program execution when a specific task is active. An example is shown on page 169.

Example

```
>_TaskRunning_ (command)       /* check if task 'command' is running    */
0001                           /* returns 1 if task is currently running*/
>_break_= _TaskRunning_ (init) /* stop program when 'init' is running   */
```

uchar _RBYTE (*address*), uint _RWORD (*address*), ulong _RDWORD (*address*), float _RFLOAT (*address*), double _RDOUBLE (*address*)

These functions return the content of the specified memory *address*.

Example

```
>_RBYTE (0x20000)             /* return the character at 0x20000  */
>_RFLOAT (0xE000)             /* return the float value at 0xE000 */
>_RDWORD (0x1000)             /* return the long value at 0x1000  */
```

_WBYTE (*addr,* uchar *value*), _WWORD (*addr,* uint *value*), _WDWORD (*addr,* ulong *value*), _WFLOAT (*addr,* float *value*, _WDOUBLE (*addr,* double *value*)

These functions write a *value* to the specified memory *address*.

Example

```
>_WBYTE (0x20000, 0x55)       /* write the byte 0x33 at 0x20000  */
>_RFLOAT (0xE000, 1.5)        /* write the float value 1.5 at 0xE000 */
>_RDWORD (0x1000, 12345678)   /* write the long value 12345678 at 0x1000*/
```

User Functions

User functions are functions you create to use with the µVision2 debugger. You may enter user functions directly in the function editor or you may use the **INCLUDE** command to load a file that contains one or more user functions.

NOTE
*µVision2 provides a number of system variables you may use in user functions. Refer to “**System Variables**” on page 101 for more information.*

User functions begin with **FUNC** keyword and are defined as follows:

```
FUNC return_type fname (parameter_list) {
  statements
}
```

return_type is the type of the value returned by the function and may be: **bit**, **char**, **float**, **int**, **long**, **uchar**, **uint**, **ulong**, **void**. You may use **void** if the function does not return a value. If no return type is specified the type **int** is assumed.

fname is the name of the function.

parameter_list is the list of arguments that are passed to the function. Each argument must have a type and a name. If no arguments are passed to the function, use **void** for the *parameter_list*. Multiple arguments are separated by commas.

statements are instructions the function carries out.

{ is the open curly brace. The function definition is complete when the number of open braces is balanced with the number of the closing braces (}).

Example

The following user function displays the contents of several CPU registers. For more information about “Creating Functions” refer to page 119.

```
FUNC void MyRegs (void)  {
  printf ("---------- MyRegs() ----------\n");
  printf (" R4   R8   R9   R10  R11  R12\n");
  printf (" %04X %04X %04X %04X %04X %04X\n",
           R4,  R8,  R9,  R10, R11, R12);
  printf ("------------------------------\n");
}
```

To invoke this function, type the following in the command window.

```
MyRegs()
```

When invoked, the **MyRegs** function displays the contents of the registers and appears similar to the following:

```
---------- MyRegs() ----------
 R4   R8   R9   R10  R11  R12
 B02C 8000 0001 0000 0000 0000
------------------------------
```

You may define a toolbox button to invoke the user function with:

```
DEFINE BUTTON "My Registers", "MyRegs()"
```

Restrictions

- µVision2 checks that the return value of a user function corresponds to the function return type. Functions with a **void** return type must not return a value. Functions with a non-**void** return type must return a value. Note that µVision2 does not check each return path for a valid return value.
- User functions may not invoke signal functions or the **twatch** function.
- The value of a local object is undefined until a value is assigned to it.
- Remove user functions using the **KILL FUNC** command.

Signal Functions

A Signal function let you repeat operations, like signal inputs and pulses, in the background while µVision2 executes your target program. Signal functions help you simulate and test serial I/O, analog I/O, port communications, and other repetitive *external* events.

Signal functions execute in the background while µVision2 simulates your target program. Therefore, a signal function must call the **twatch** function at some point to delay and let µVision2 run your target program. µVision2 reports an error for signal functions that never call **twatch**.

NOTE
*µVision2 provides a number of system variables you may use in your signal functions. Refer to "**System Variables**" on page 101 for more information.*

Signal functions begin with the **SIGNAL** keyword and are defined as follows:

```
SIGNAL void fname (parameter_list) {
  statements
}
```

fname is the name of the function.

parameter_list is the list of arguments that are passed to the function. Each argument must have a type and a name. If no arguments are passed to the function, use **void** for the *parameter_list*. Multiple arguments are separated by commas.

statements are instructions the function carries out.

{ is the open curly brace. The function definition is complete when the number of open braces is balanced with the number of the closing braces ("}").

Example

The following example shows a signal function that puts the character 'A' into the serial input buffer once every 1,000,000 CPU states. For more information about "Creating Functions" refer to page 119.

```
SIGNAL void StuffS0in (void)  {
  while (1) {
    S0IN = 'A';
    twatch (1000000);
  }
}
```

To invoke this function, type the following in the command window.

```
StuffS0in()
```

When invoked, the **StuffS0in** signal function puts and ASCII character 'A' in the serial input buffer, delays for 1,000,000 CPU states, and repeats.

Restrictions

The following restrictions apply to signal functions:

- The return type of a signal function must be **void**.
- A signal function may have a maximum of eight function parameters.
- A signal function may invoke other predefined functions and user functions.
- A signal function may not invoke another signal function.
- A signal function may be invoked by a user function.
- A signal function must call the **twatch** function at least once. Signal functions that never call **twatch** do not allow the target program time to execute. Since you cannot use **Ctrl+C** to abort a signal function, µVision2 may enter an infinite loop.

Managing Signal Functions

µVision2 maintains a queue for active signal functions. A signal function may either be either idle or running. A signal function that is idle is delayed while it waits for the number of CPU states specified in a call to **twatch** to expire. A signal function that is running is executing statements inside the function.

When you invoke a signal function, µVision2 adds that function to the queue and marks it as running. Signal functions may only be activated once, if the function is already in the queue, a warning is displayed. View the state of active signal functions with the command **SIGNAL STATE**. Remove active signal functions form the queue with the command **SIGNAL KILL**.

When a signal function invokes the **twatch** function, it goes in the idle state for the number of CPU states passed to **twatch**. After the user program has executed the specified number of CPU states, the signal function becomes running. Execution continues at the statement after **twatch**.

If a signal function exits, because of a return statement, it is automatically removed from the queue of active signal functions.

Analog Example

The following example shows a signal function that varies the input to analog input 0 on a C167. The function increases and decreases the input voltage by 0.5 volts from 0V and an upper limit that is specified as the signal function’s only argument. This signal function repeats indefinitely, delaying 200,000 states for each voltage step.

```
signal void analog0 (float limit)  {
  float volts;

  printf ("Analog0 (%f) entered.\n", limit);
  while (1)  {          /* forever */
    volts = 0;
    while (volts <= limit)  {
      ain0 = volts;     /* analog input-0 */
      twatch (200000);  /* 200000 states Time-Break */
      volts += 0.1;     /* increase voltage */
    }
    volts = limit;
    while (volts >= 0.0)  {
      ain0 = volts;
      twatch (200000);  /* 200000 states Time-Break */
      volts -= 0.1;     /* decrease voltage */
    }
  }
}
```

The signal function **analog0** can then be invoked as follows:

```
>ANALOG0 (5.0)                                    /* Start of 'ANALOG()' */
ANALOG0 (5.000000) ENTERED
```

The **SIGNAL STATE** command to displays the current state of the **analog0**:

```
>SIGNAL STATE
 1  idle     Signal = ANALOG0 (line 8)
```

µVision2 lists the internal function number, the status of the signal function: idle or running, the function name and the line number that is executing.

Since the status of the signal function is idle, you can infer that **analog0** executed the **twatch** function (on line 8 of **analog0**) and is waiting for the specified number of CPU states to elapse. When 200,000 states pass, **analog0** continues execution until the next call to **twatch** in line 8 or line 14.

The following command removes the **analog0** signal function from the queue of active signal functions.

```
>SIGNAL KILL ANALOG0
```

Differences Between Debug Functions and C

There are a number of differences between ANSI C and the subset of features support in µVision2 debug user and signal functions.

- µVision2 does not differentiate between uppercase and lowercase. The names of objects and control statements may be written in either uppercase or lowercase.
- µVision2 has no preprocessor. Preprocessor directives like **#define**, **#include**, and **#ifdef** are not supported.
- µVision2 does not support global declarations. Scalar variables must be declared within a function definition. You may define symbols with the **DEFINE** command and use them like you would use a global variable.
- In µVision2, variables may not be initialized when they are declared. Explicit assignment statements must be used to initialize variables.
- µVision2 functions only support scalar variable types. Structures, arrays, and pointers are not allowed. This applies to the function return type as well as the function parameters.
- µVision2 functions may only return scalar variable types. Pointers and structures may not be returned.
- µVision2 functions cannot be called recursively. During function execution, µVision2 recognizes recursive calls and aborts function execution if one is detected.
- µVision2 functions may only be invoked directly using the function name. Indirect function calls via pointers are not supported.
- µVision2 supports only the ANSI style for function declarations with a parameter list. The old K&R format is not supported. For example, the following ANSI style function is acceptable.

```
func test (int pa1, int pa2)  {                 /* ANSI type, correct */
  /* ...  */
}
```

The following K&R style function is not acceptable.

```
func test (pa1, pa2)                               /* Old K&R style is */
int pa1, pa2;                                         /* not supported */
{
  /* ... */
}
```

Differences Between µVision2 and dScope

The µVision2 debugger replaces the Keil dScope for Windows. dScope debug functions require the following modifications for correct execution in the µVision2 debugger.

- In dScope the **memset** debug function parameters are different. The µVision2 **memset** debug function parameters are now identical with the ANSI C **memset** function.
- The dScope debug function **bit** is no longer available and needs to be replaced with **_RBYTE** and **_WBYTE** function calls. With dScope debug functions **char**, **uchar**, **int**, **uint**, **long**, **ulong**, **float**, and **double** it is possible to read and write memory. Replace these debug functions in µVision2 according the following list.

dScope Memory	µVision2 Replacement for	
Access Function	Memory Read	Memory Write
bit	_RBYTE	combine _RBYTE and _WBYTE
char, uchar	_RBYTE	_WBYTE
int, uint	_RWORD	_WWORD
long, ulong	_RDWORD	_WDWORD
float	_RFLOAT	_WFLOAT
double	_RDOUBLE	_WDOUBLE

- In dScope the **memset** debug function parameters are different. The µVision2 **memset** debug function parameters are now identical with the ANSI C **memset** function.

Chapter 7. Sample Programs

This section describes the sample programs that are included in our tool kits. The sample programs are ready for you to run. You can use the sample programs to learn how to use our tools. Additionally, you can copy the code from our samples for your own use.

The sample programs are found in the **C:\KEIL\C51\EXAMPLES\** folder. Each sample program is stored in a separate folder along with project files that help you quickly build and evaluate each sample program.

The following table lists the sample programs and their folder names.

Example	Description
BADCODE	Program with syntax errors and warnings. You may use the µVision2 editor to correct these.
CSAMPLE	Simple addition and subtraction calculator that shows how to build a multi-module project with µVision2.
DHRY	Dhrystone benchmark. Calculates the dhrystones factor for the target CPU.
HELLO	Hello World program. Try this first when you begin using µVision2. It prints Hello World on the serial interface and helps you confirm that the development tools work correctly. Refer to “HELLO: Your First 8051 C Program” on page 142 for more information about this sample program.
MEASURE	Data acquisition system that collects analog and digital signals. Refer to “MEASURE: A Remote Measurement System” on page 146 for more information about this sample program.
RTX_EX1	Demonstrates round-robin multitasking using RTX-51 Tiny.
RTX_EX2	Demonstrates an RTX-51 Tiny application that uses signals.
SIEVE	Benchmark that calculates prime numbers.
TRAFFIC	Shows how to control a traffic light using the RTX-51 Tiny real-time executive.
WHETS	Benchmark program that calculates the whetstones factor for the target CPU.

To begin using one of the sample projects, use the µVision2 menu **Project – Open Project** and load the project file.

The following sections in this chapter describe how to use the tools to build the following sample programs:

- HELLO: Your First 8051 C Program
- MEASURE: A Remote Measurement System

HELLO: Your First 8051 C Program

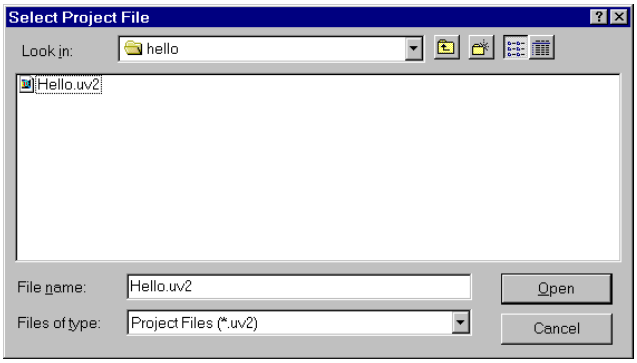
The HELLO sample program is located in **C:\KEIL\C51\EXAMPLES\HELLO\** . HELLO does nothing more than print the text “Hello World” to the serial port. The entire program is contained in a single source file **HELLO.C.**

This small application helps you confirm that you can compile, link, and debug an application. You can perform these operations from the DOS command line, using batch files, or from µVision2 for Windows using the provided project file.

The hardware for HELLO is based on the standard 8051 CPU. The only on-chip peripheral used is the serial port. You do not actually need a target CPU because µVision2 lets you simulate the hardware required for this program.

HELLO Project File

In µVision, applications are maintained in a project file. A project file has been created for HELLO. To load this project, select **Open Project** from the **Project** menu and open **HELLO.UV2** from the folder **...\C51\EXAMPLES\HELLO**.



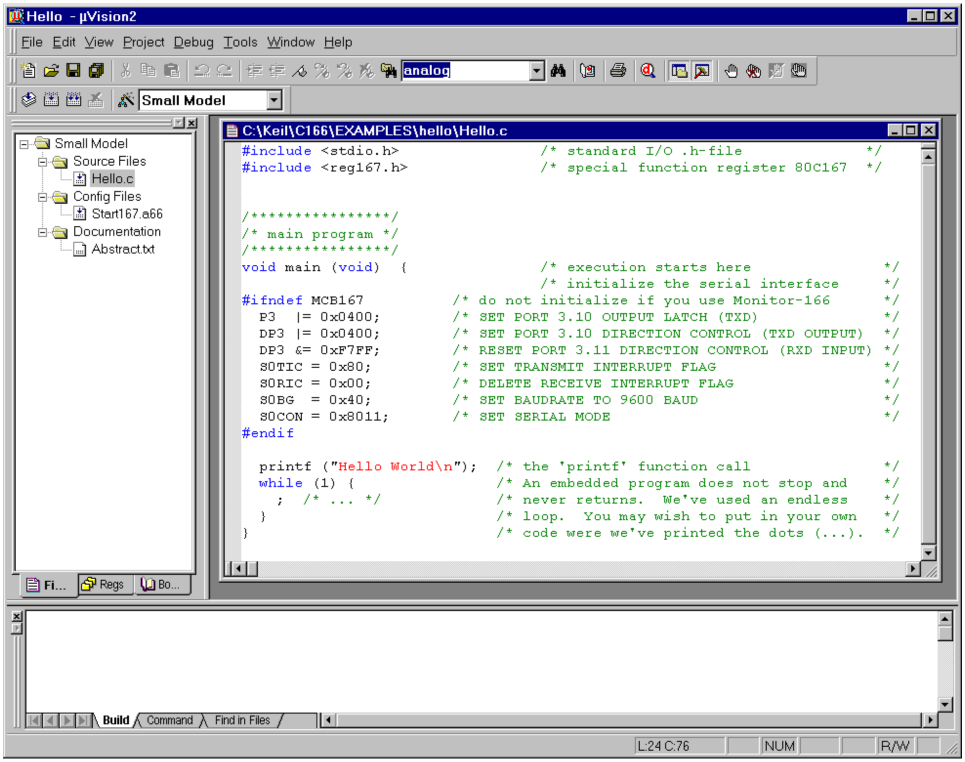
Look in: hello

File name: Hello.uv2

Files of type: Project Files (*.uv2)

Editing HELLO.C

You can now edit **HELLO.C**. Double click on **HELLO.C** in the Files page of the Project Window. µVision2 loads and displays the contents of **HELLO.C** in an editor window.



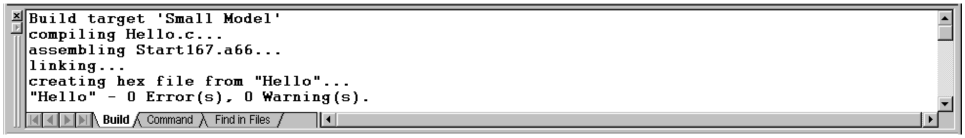
```
#include <stdio.h>                /* standard I/O .h-file               */
#include <reg167.h>               /* special function register 80C167   */


/****************/
/* main program */
/****************/
void main (void)  {               /* execution starts here                */
                                  /* initialize the serial interface      */
#ifndef MCB167          /* do not initialize if you use Monitor-166       */
  P3  |= 0x0400;        /* SET PORT 3.10 OUTPUT LATCH (TXD)               */
  DP3 |= 0x0400;        /* SET PORT 3.10 DIRECTION CONTROL (TXD OUTPUT)   */
  DP3 &= 0xF7FF;        /* RESET PORT 3.11 DIRECTION CONTROL (RXD INPUT)  */
  S0TIC = 0x80;         /* SET TRANSMIT INTERRUPT FLAG                    */
  S0RIC = 0x00;         /* DELETE RECEIVE INTERRUPT FLAG                  */
  S0BG  = 0x40;         /* SET BAUDRATE TO 9600 BAUD                      */
  S0CON = 0x8011;       /* SET SERIAL MODE                                */
#endif

  printf ("Hello World\n");  /* the 'printf' function call                */
  while (1) {                /* An embedded program does not stop and     */
    ;  /* ... */             /* never returns.  We've used an endless     */
  }                          /* loop.  You may wish to put in your own    */
}                            /* code were we've printed the dots (...).   */
```

Compiling and Linking HELLO

When you are ready to compile and link your project, use the **Build Target** command from the **Project** menu or the Build toolbar. µVision2 begins to translate and link the source files and creates an absolute object module that you can load into the µVision2 debugger for testing. The status of the build process is listed in the **Build** page of the **Output Window**.



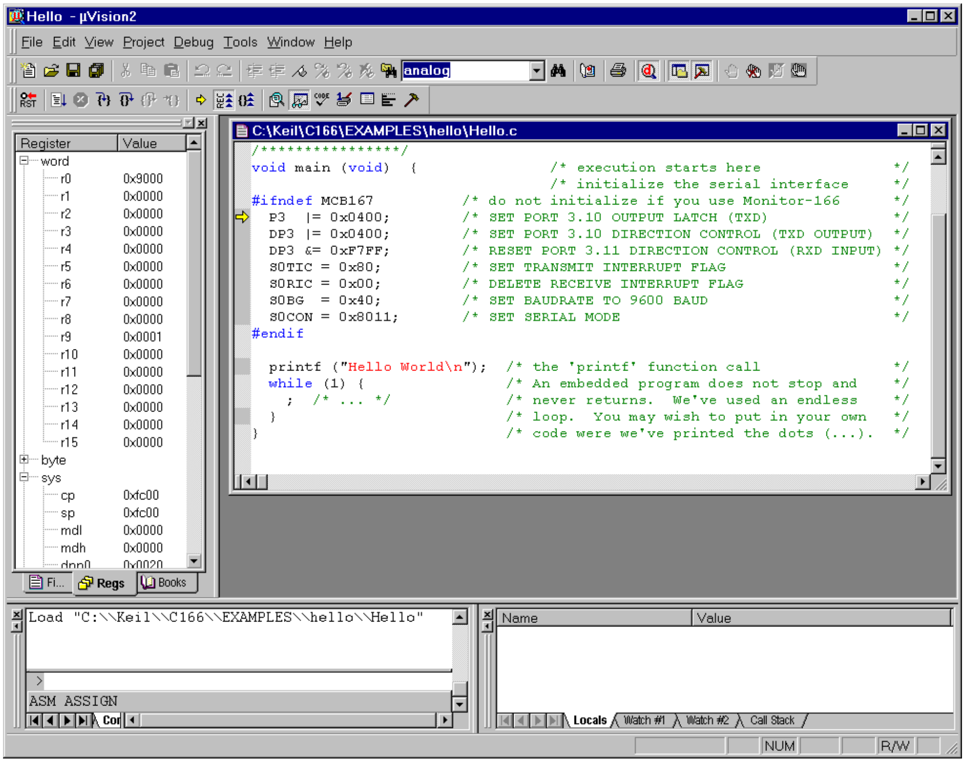
```
Build target 'Small Model'
compiling Hello.c...
assembling Start167.a66...
linking...
creating hex file from "Hello"...
"Hello" - 0 Error(s), 0 Warning(s).
```

NOTE

You should encounter no errors when you use µVision2 with the provided sample projects.

Testing HELLO

Once the HELLO program is compiled and linked, you can test it with the µVision2 debugger. In µVision2, use the **Start/Stop Debug Session** command from the **Debug** menu or toolbar. µVision2 initializes the debugger and starts program execution till the main function. The following screen displays.


Hello - µVision2
File Edit View Project Debug Tools Window Help
analog
C:\Keil\C166\EXAMPLES\hello\Hello.c
Fi...
Regs
Books
Cor
Locals
Watch #1
Watch #2
Call Stack
NUM
R/W

Register	Value
word	
r0	0x9000
r1	0x0000
r2	0x0000
r3	0x0000
r4	0x0000
r5	0x0000
r6	0x0000
r7	0x0000
r8	0x0000
r9	0x0001
r10	0x0000
r11	0x0000
r12	0x0000
r13	0x0000
r14	0x0000
r15	0x0000
byte	
sys	
cp	0xfc00
sp	0xfc00
mdl	0x0000
mdh	0x0000
dpp0	0x0020

```
/*****************/
void main (void)  {                    /* execution starts here           */
                                       /* initialize the serial interface */
#ifndef MCB167              /* do not initialize if you use Monitor-166   */
  P3   |= 0x0400;           /* SET PORT 3.10 OUTPUT LATCH (TXD)           */
  DP3  |= 0x0400;           /* SET PORT 3.10 DIRECTION CONTROL (TXD OUTPUT) */
  DP3  &= 0xF7FF;           /* RESET PORT 3.11 DIRECTION CONTROL (RXD INPUT) */
  S0TIC = 0x80;             /* SET TRANSMIT INTERRUPT FLAG                 */
  S0RIC = 0x00;             /* DELETE RECEIVE INTERRUPT FLAG               */
  S0BG  = 0x40;             /* SET BAUDRATE TO 9600 BAUD                   */
  S0CON = 0x8011;           /* SET SERIAL MODE                             */
#endif

  printf ("Hello World\n");  /* the 'printf' function call               */
  while (1) {                /* An embedded program does not stop and    */
    ;  /* ... */             /* never returns.  We've used an endless    */
  }                          /* loop.  You may wish to put in your own   */
}                            /* code were we've printed the dots (...).  */
```

```
Load "C:\\Keil\\C166\\EXAMPLES\\hello\\Hello"
>
ASM ASSIGN
```

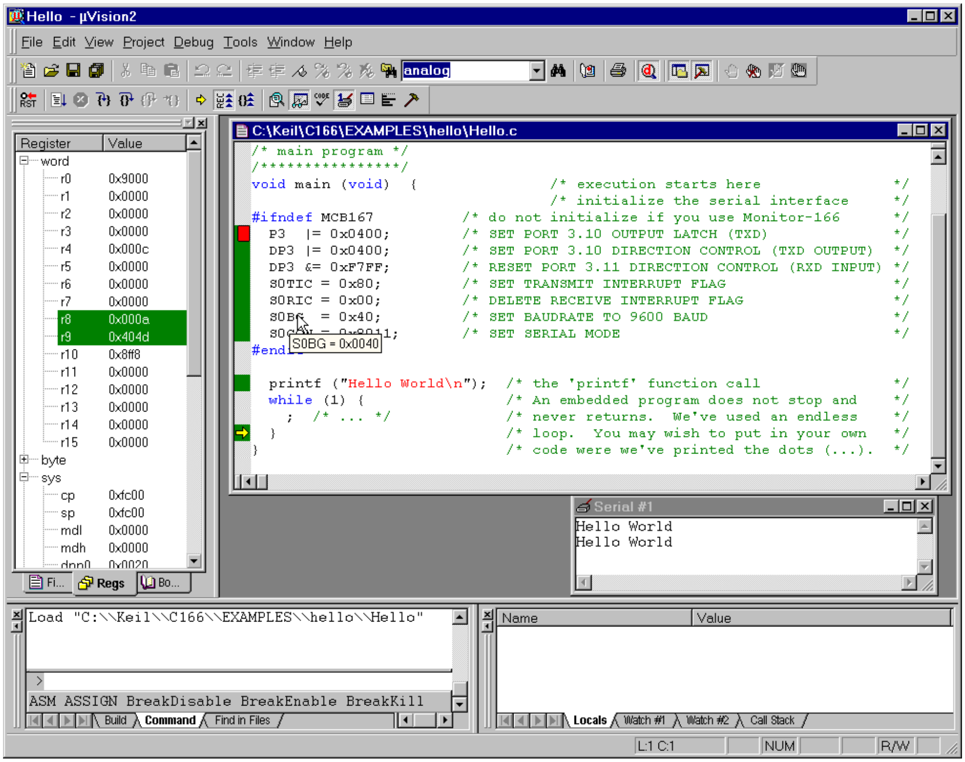
Name	Value

Open **Serial Window #1** that displays the serial output of the application with the **Serial Window #1** command from the **View** menu or the **Debug** toolbar.

Run HELLO with the **Go** command from the **Debug** menu or toolbar. The HELLO program executes and displays the text “Hello World” in the serial window. After HELLO outputs “Hello World,” it begins executing an endless loop.

Stop Running HELLO with the **Halt** command from the **Debug** menu or the toolbar. You may also type **ESC** in the Command page of the Output window.

During debugging µVision2 will show the following output:


Hello - µVision2
File Edit View Project Debug Tools Window Help
analog
RST
Fi... Regs Bo...
Build Command Find in Files
Locals Watch #1 Watch #2 Call Stack
L1 C:1 NUM R/W

C:\Keil\C166\EXAMPLES\hello\Hello.c

```
/* main program */
/****************/
void main (void)  {                  /* execution starts here           */
                                     /* initialize the serial interface */
#ifndef MCB167             /* do not initialize if you use Monitor-166        */
  P3  |= 0x0400;           /* SET PORT 3.10 OUTPUT LATCH (TXD)                */
  DP3 |= 0x0400;           /* SET PORT 3.10 DIRECTION CONTROL (TXD OUTPUT)    */
  DP3 &= 0xF7FF;           /* RESET PORT 3.11 DIRECTION CONTROL (RXD INPUT)   */
  S0TIC = 0x80;            /* SET TRANSMIT INTERRUPT FLAG                     */
  S0RIC = 0x00;            /* DELETE RECEIVE INTERRUPT FLAG                   */
  S0BG  = 0x40;            /* SET BAUDRATE TO 9600 BAUD                       */
  S0CON = 0x8011;          /* SET SERIAL MODE                                 */
#endif

  printf ("Hello World\n");  /* the 'printf' function call              */
  while (1) {                /* An embedded program does not stop and   */
    ;  /* ... */             /* never returns.  We've used an endless   */
  }                          /* loop.  You may wish to put in your own  */
}                            /* code were we've printed the dots (...). */
```

S0BG = 0x0040

Register	Value
word	
r0	0x9000
r1	0x0000
r2	0x0000
r3	0x0000
r4	0x000c
r5	0x0000
r6	0x0000
r7	0x0000
r8	0x000a
r9	0x404d
r10	0x8ff8
r11	0x0000
r12	0x0000
r13	0x0000
r14	0x0000
r15	0x0000
byte	
sys	
cp	0xfc00
sp	0xfc00
mdl	0x0000
mdh	0x0000
dpp0	0x0020

Serial #1

```
Hello World
Hello World
```

```
Load "C:\\Keil\\C166\\EXAMPLES\\hello\\Hello"
>
ASM ASSIGN BreakDisable BreakEnable BreakKill
```

Name	Value

Single-Stepping and Breakpoints

Use the **Insert/Remove Breakpoints** command from the toolbar or the local editor menu that opens with a right mouse click and set a breakpoint at the beginning of the main function.

Use the **Reset CPU** command from the Debug menu or toolbar. If you have halted HELLO start program execution with **Run**. µVision2 will stop the program at the breakpoint.

You can single-step through the HELLO program using the Step buttons in the debug toolbar. The current instruction is marked with a yellow arrow. The arrow moves each time you step

Place the mouse cursor over a variable to view their value.

You may stop debugging at any time with **Start/Stop Debug Session** command.

MEASURE: A Remote Measurement System

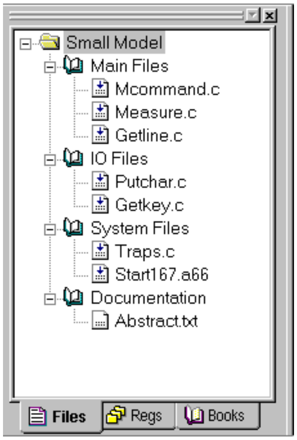
The MEASURE sample program is located in the **\C51\EXAMPLES\MEASURE\** folder. MEASURE runs a remote measurement system that collects analog and digital data like a data acquisition systems found in a weather stations and process control applications. MEASURE is composed of three source files: **GETLINE.C**, **MCOMMAND.C**, and **MEASURE.C**.

This implementation records data from two digital ports and four A/D inputs. A timer controls the sample rate. The sample interval can be configured from 1 millisecond to 60 minutes. Each measurement saves the current time and all of the input channels to a RAM buffer.

Hardware Requirements

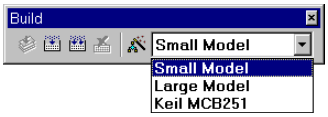
The hardware for MEASURE is based on the C515 CPU. This microcontroller provides analog and digital input capability. Port 4 and port 5 is used for the digital inputs and AN0 through AN3 are used for the analog inputs. You do not actually need a target CPU because µVision2 lets you simulate all the hardware required for this program.

MEASURE Project File



The project file for the MEASURE sample program is called **MEASURE.UV2**. To load this project file, use **Open Project** from the **Project** menu and select **MEASURE.UV2** in the folder **C:\KEIL\C51\EXAMPLES\MEASURE**.

The Files page in the Project Window shows the source files that compose the MEASURE project. The three application related source files that are located in the **Main Files** group. The function of the source files is described below. To open a source file, double-click on the filename.



The project contains several targets for different test environments. For debugging with the simulator select the target Small Model in the Build toolbar.

MEASURE.C contains the main C function for the measurement system and the interrupt routine for timer 0. The main function initializes all peripherals of the C515 and performs command processing for the system. The timer interrupt routine, timer0, manages the real-time clock and the measurement sampling of the system.

MCOMMAND.C processes the display, time, and interval commands. These functions are called from main. The display command lists the analog values in floating-point format to give a voltage between 0.00V and 5.00V.

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

Compiling and Linking MEASURE

When you are ready to compile and link MEASURE, use the **Build Target** command from the Project menu or the toolbar. µVision2 begins to compile and link the source files in MEASURE and displays a message when the build is finished.

Once the project is build, you are ready to browse the symbol information or begin testing the MEASURE program.

Browse Symbols

The MEASURE project is configured to generate full browse and debug information. To view the information, use the **Source Browse** command from the View menu or the toolbar. For more information refer to “Source Browser” on page 58.

Testing MEASURE

The MEASURE sample program is designed to accept commands from the on-chip serial port. If you have actual target hardware, you can use a terminal simulation to communicate with the C515 CPU. If you do not have target hardware, you can use µVision2 to simulate the hardware. You can also use the serial window in µVision2 to provide serial input.

Once the MEASURE program is build, you can test it. Use the **Start/Stop Debug Session** command from the **Debug** menu to start the µVision2 debugger.

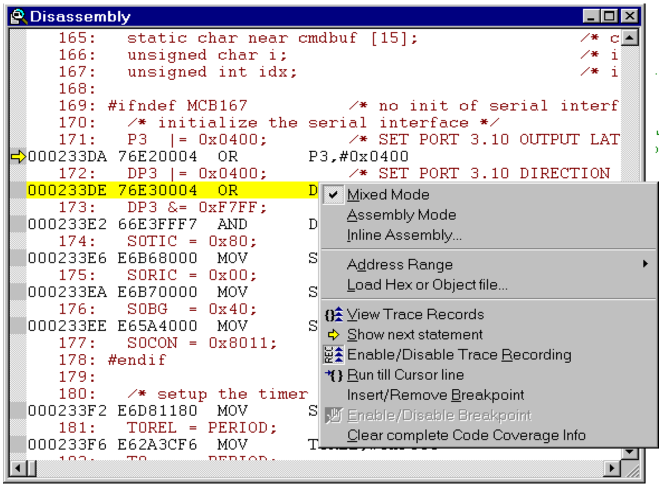
Remote Measurement System Commands

The serial commands that MEASURE supports are listed in the following table. These commands are composed of ASCII text characters. All commands must be terminated with a carriage return. You can enter these commands in the **Serial Window #1** during debugging.

Command	Serial Text	Description
Clear	C	Clears the measurement record buffer.
Display	D	Displays the current time and input values.
Time	T *hh:mm:ss*	Sets the current time in 24-hour format.
Interval	I *mm:ss.ttt*	Sets the interval time for the measurement samples. The interval time must be between 0:00.001 (for 1ms) and 60:00.000 (for 60 minutes).
Start	S	Starts the measurement recording. After receiving the start command, MEASURE samples all data inputs at the specified interval.
Read	R [*count*]	Displays the recorded measurements. You may specify the number of most recent samples to display with the read command. If no count is specified, the read command transmits all recorded measurements. You can read measurements on the fly if the interval time is more than 1 second. Otherwise, the recording must be stopped.
Quit	Q	Quits the measurement recording.

View Program Code

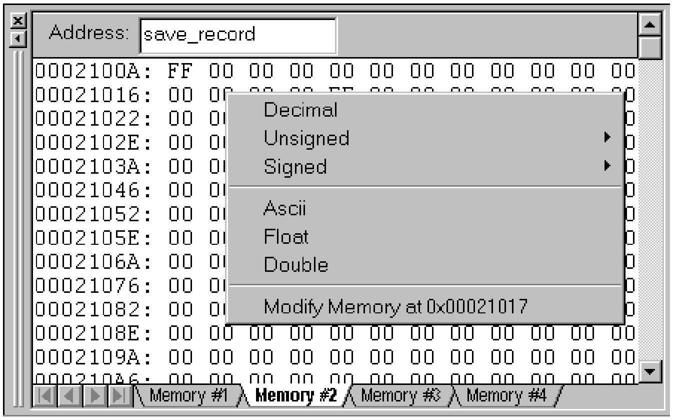
µVision2 lets you view the program code in the Disassembly Window that opens with the View menu or the toolbar button. The Disassembly Window shows intermixed source and assembly lines. You may change the view mode or use other commands from the local menu that opens with the right mouse button.



```
   165:   static char near cmdbuf [15];                 /* c
   166:   unsigned char i;                              /* i
   167:   unsigned int idx;                             /* i
   168:
   169: #ifndef MCB167              /* no init of serial interf
   170:   /* initialize the serial interface */
   171:   P3  |= 0x0400;            /* SET PORT 3.10 OUTPUT LAT
000233DA 76E20004  OR         P3,#0x0400
   172:   DP3 |= 0x0400;            /* SET PORT 3.10 DIRECTION
000233DE 76E30004  OR         D
   173:   DP3 &= 0xF7FF;
000233E2 66E3FFF7  AND        D
   174:   S0TIC = 0x80;
000233E6 E6B68000  MOV        S
   175:   S0RIC = 0x00;
000233EA E6B70000  MOV        S
   176:   S0BG  = 0x40;
000233EE E65A4000  MOV        S
   177:   S0CON = 0x8011;
   178: #endif
   179:
   180:   /* setup the timer
000233F2 E6D81180  MOV        S
   181:   T0REL = PERIOD;
000233F6 E62A3CF6  MOV        T
```

View Memory Contents

µVision2 displays memory in various formats. The Memory Window opens via the View menu or the toolbar button. You can enter the address of four different memory areas in the pages. The local menu allows you to modify the memory contents or select different output formats.



Address: save_record

```
0002100A: FF 00 00 00 00 00 00 00 00 00 00 00
00021016: 00 00
00021022: 00 0
0002102E: 00 0
0002103A: 00 0
00021046: 00 0
00021052: 00 0
0002105E: 00 0
0002106A: 00 0
00021076: 00 0
00021082: 00 0
0002108E: 00 00 00 00 00 00 00 00 00 00 00 00
0002109A: 00 00 00 00 00 00 00 00 00 00 00 00
000210A6: 00 00 00 00 00 00 00 00 00 00 00 00
```

Program Execution

Before you begin simulating MEASURE, open the **Serial Window #1** that displays the serial output with the **View** menu or the **Debug** toolbar. You may disable other windows if your screen is not large enough.

You can use the Step toolbar buttons on assembler instructions or source code lines. If the Disassembly Window is active, you single step at assembly instruction basis. If an editor window with source code is active, you single step at source code level.

The **StepInto** toolbar button lets you single-step through your application and into function calls.

StepOver executes a function call as single entity and is not interrupt unless a breakpoint occurs.

On occasion, you may accidentally step into a function unnecessarily. You can use **StepOut** to complete execution of that function and return to the statement immediately following the function call.

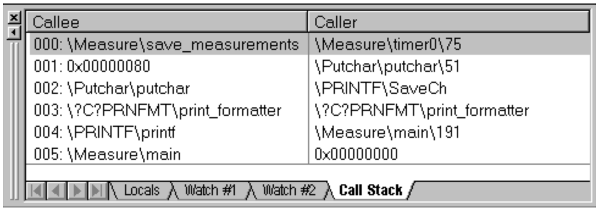
A yellow arrow marks the current assembly or high-level statement. You may use the you may accidentally step into a function unnecessarily. You can use **StepOut** to complete execution of that function and return to the statement immediately following the function call.

The toolbar or local menu command **Run till Cursor Line** lets you use the current cursor line as temporary breakpoint.

With **Insert/Remove Breakpoints** command you can set or remove breakpoints on high-level source lines or assembler statements.

Call Stack

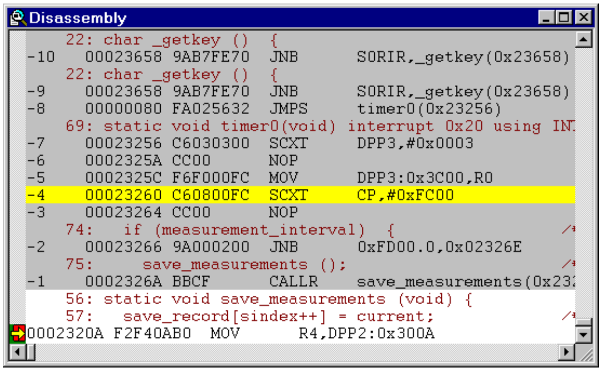
µVision2 internally tracks function nesting as the program executes. The **Call Stack** page of the **Watch Window** shows the current function nesting. A double click on a line displays the source code that called the selected function.



Callee	Caller
000: \Measure\save_measurements	\Measure\timer0\75
001: 0x00000080	\Putchar\putchar\51
002: \Putchar\putchar	\PRINTF\SaveCh
003: \?C?PRNFMT\print_formatter	\?C?PRNFMT\print_formatter
004: \PRINTF\printf	\Measure\main\191
005: \Measure\main	0x00000000

Trace Recording

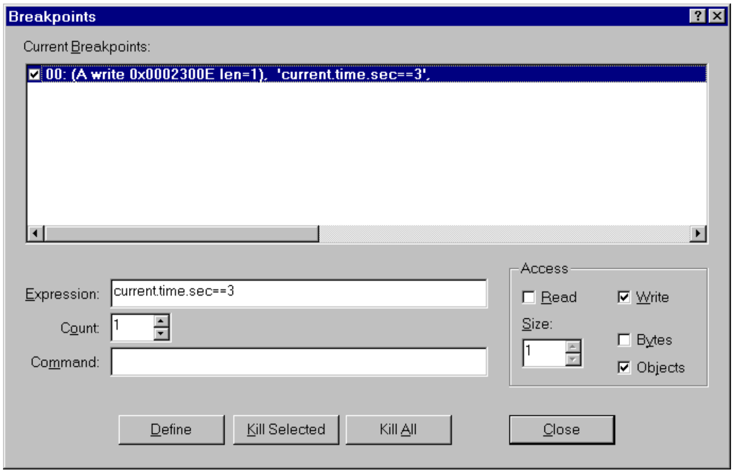
It is common during debugging to reach a breakpoint where you require information like register values and other circumstances that led to the breakpoint. If **Enable/Disable Trace Recording** is set you can view the CPU instructions that were executed be reaching the breakpoint. The **Regs** page of the **Project Window** shows the CPU register contents for the selected instruction.



```
     22: char _getkey ()  {
-10    00023658 9AB7FE70  JNB      S0RIR,_getkey(0x23658)
     22: char _getkey ()  {
-9     00023658 9AB7FE70  JNB      S0RIR,_getkey(0x23658)
-8     00000080 FA025632  JMPS     timer0(0x23256)
     69: static void timer0(void) interrupt 0x20 using IN
-7     00023256 C6030300  SCXT     DPP3,#0x0003
-6     0002325A CC00      NOP
-5     0002325C F6F000FC  MOV      DPP3:0x3C00,R0
-4     00023260 C60800FC  SCXT     CP,#0xFC00
-3     00023264 CC00      NOP
     74:   if (measurement_interval)  {                  /*
-2     00023266 9A000200  JNB      0xFD00.0,0x02326E
     75:      save_measurements ();                      /*
-1     0002326A BBCF      CALLR    save_measurements(0x23
     56: static void save_measurements (void) {
     57:   save_record[sindex++] = current;              /*
0002320A F2F40AB0  MOV      R4,DPP2:0x300A
```

Breakpoints Dialog

µVision2 also supports complex breakpoints as discussed on page 83. You may want to halt program execution when a variable contains a certain value. The example shows how to stop when the value 3 is written to **current.time.sec**.



Breakpoints

Current Breakpoints:

☑ 00: (A write 0x0002300E len=1), 'current.time.sec==3',

Expression: current.time.sec==3

Count: 1

Command:

Access: □ Read ☑ Write

Size: 1

□ Bytes ☑ Objects

Open the **Breakpoints** dialog from the **Debug** menu. Enter as expression `current.time.sec==3`. Select the Write check box (this option specifies that the break condition is tested only when the expression is written to). Click on the Define button to set the breakpoint.

To test the breakpoint condition perform the following steps:

Reset CPU.

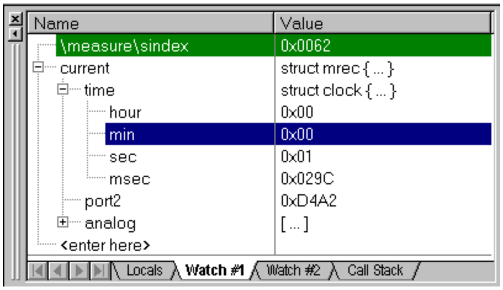
If program execution is halted begin executing the MEASURE program.

After a few seconds, µVision2 halts execution. The program counter line in the debug window marks the line in which the breakpoint occurred.

Watch Variables

You may constantly view the contents of variables, structures, and arrays. Open the **Watch Window** from the View menu or with the toolbar. The **Locals** page shows all local symbols of the current function. The Watch #1 and Watch #2 pages allow you to enter any program variables as described in the following:

- Select the text **<enter here>** with a mouse click and wait a second. Another mouse click enters edit mode that allows you to add variables. In the same way you can modify variable values.
- Select a variable name in an **Editor Window** and open the local menu with a right mouse click and use the command **Add to Watch Window.**
- You can enter **WatchSet** in the **Output Window – Command** page.



Name	Value
\measure\sindex	0x0062
⊟ current	struct mrec {...}
⊟ time	struct clock {...}
hour	0x00
min	0x00
sec	0x01
msec	0x029C
port2	0xD4A2
⊞ analog	[...]
<enter here>	

To remove a variable, click on the line and press the **Delete** key.

Structures and arrays open on demand when you click on the [+] symbol. Display lines are indented to reflect the nesting level.

The Watch Window updates at the end of each execution command. You enable may enable **Periodic Window Update** in the **View** menu to update the watch window during program execution.

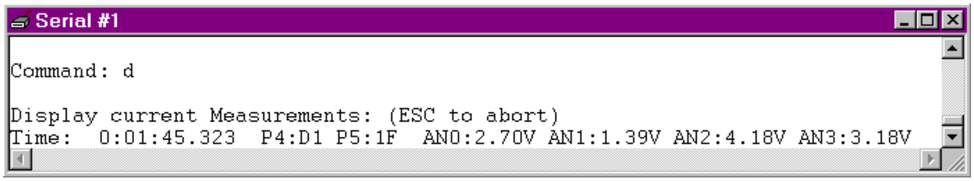
View and Modify On-Chip Peripherals

µVision2 provides several ways to view and modify the on-chip peripherals used in your target program. You may directly view the results of the example below when you perform the following steps:

Reset CPU and kill all defined breakpoints.

If program execution is halted begin executing the MEASURE program.

Open the **Serial Window #1** and enter the ‘d’ command for the MEASURE application. MEASURE shows the values from I/O Port2 and A/D input 0 – 3. The Serial Window shows the following output:



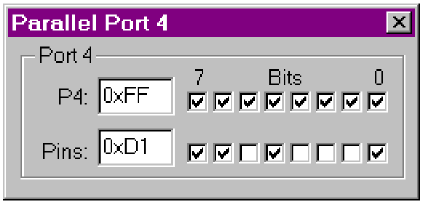
```
Command: d

Display current Measurements: (ESC to abort)
Time:  0:01:45.323  P4:D1 P5:1F  AN0:2.70V AN1:1.39V AN2:4.18V AN3:3.18V
```

You may now use the following procedures to supply input to the I/O pins:

Using Peripheral Dialog Boxes

µVision2 provides dialogs for: I/O Ports, Interrupts, Timers, A/D Converter, Serial Ports, and chip-specific peripherals. These dialogs can be opened from the Debug menu. For the MEASURE application you may open I/O Ports:Port4 and A/D Converter. The dialogs show the current status of the peripherals and you may directly change the input values.



Parallel Port 4

Port 4

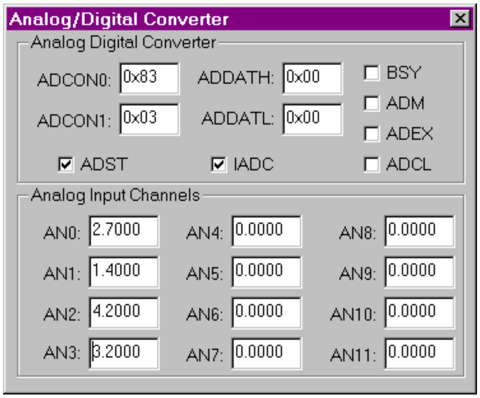
P4: 0xFF

Bits 7 … 0: ☑ ☑ ☑ ☑ ☑ ☑ ☑ ☑

Pins: 0xD1

☑ ☑ □ ☑ □ □ □ ☑

Each of these dialogs lists the related SFR symbols and shows the current status of the peripherals. To change the inputs, change the values of the Pins or Analog Input Channels.



Analog/Digital Converter

Analog Digital Converter

ADCON0: 0x83 ADDATH: 0x00 □ BSY

ADCON1: 0x03 ADDATL: 0x00 □ ADM

□ ADEX

☑ ADST ☑ IADC □ ADCL

Analog Input Channels

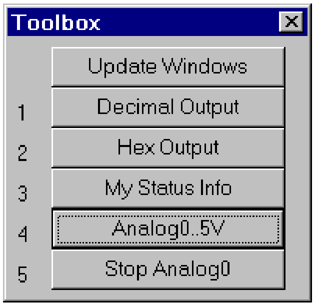
AN0:	2.7000	AN4:	0.0000	AN8:	0.0000
AN1:	1.4000	AN5:	0.0000	AN9:	0.0000
AN2:	4.2000	AN6:	0.0000	AN10:	0.0000
AN3:	3.2000	AN7:	0.0000	AN11:	0.0000

Using VTREG Symbols

You may use the "CPU Pin Registers (VTREGs)" described on page 102 to change input signals. In the **Command** page of the **Output Window**, you may make assignments to the VTREG symbols just like variables and registers. For example:

```
PORT4=0xDA                                  set digital input PORT4 to 0xDA.
AIN1=3.3                                  set analog input AIN1 to 3.3 volts.
```

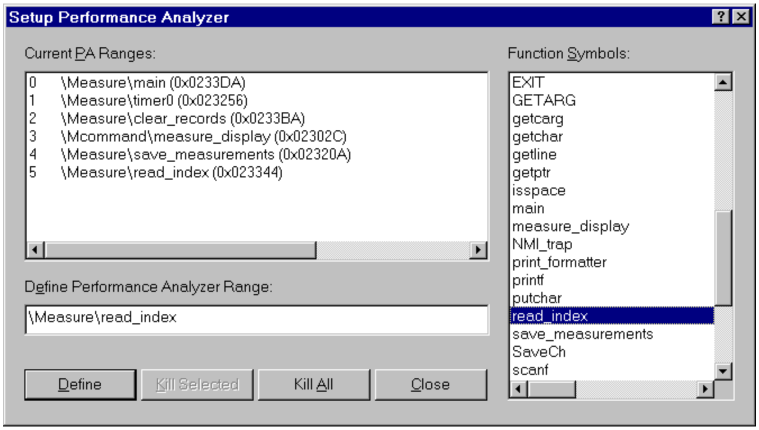
Using User and Signal Functions



You may combine the use of VTREG symbols defined by the CPU driver and µVision2 user and signal functions to create a sophisticated method of providing external input to your target programs. The “Analog Example” on page 136 shows a signal function that provides input to AIN0. The signal function is included in the MEASURE example and may be quickly invoked with the Toolbox button **Analog0..5V** and changes constantly the voltage on the input AIN0.

Using the Performance Analyzer

µVision2 lets you perform timing analysis of your applications using the integrated performance analyzer. To prepare for timing analysis, halt program execution and open the Setup **Performance Analyzer** dialog with the **Debug** menu.



```
0   \Measure\main (0x0233DA)
1   \Measure\timer0 (0x023256)
2   \Measure\clear_records (0x0233BA)
3   \Mcommand\measure_display (0x02302C)
4   \Measure\save_measurements (0x02320A)
5   \Measure\read_index (0x023344)
```

Define Performance Analyzer Range: \Measure\read_index

Function Symbols:

```
EXIT
GETARG
getcarg
getchar
getline
getptr
isspace
main
measure_display
NMI_trap
print_formatter
printf
putchar
read_index
save_measurements
SaveCh
scanf
```

You may specify the function names dialog box available from the Setup menu.

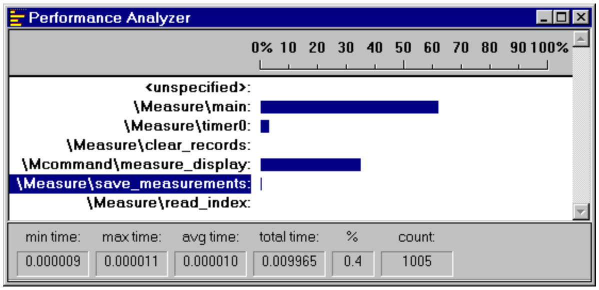
Perform the following steps to see the performance analyzer in action:

Open the Performance Analyzer using the **View** menu or toolbar.

Reset CPU and kill all breakpoints.

If program execution is halted begin executing the MEASURE program.

Select the **Serial Window #1** and type the commands **S Enter D Enter**



<unspecified>:
\Measure\main:
\Measure\timer0:
\Measure\clear_records:
\Mcommand\measure_display:
\Measure\save_measurements:
\Measure\read_index:

min time:	max time:	avg time:	total time:	%	count:
0.000009	0.000011	0.000010	0.009965	0.4	1005

The Performance Analyzer shows a bar graph for each range. The bar graph shows the percent of the time spent executing code in each range. Click on the range to see detailed timing statistics. Refer to page 91 for more information.

The MEASURE application may be also tested on a Keil MCB517 board or other C515 or C517 starter kits.

Chapter 8. RTX-51 Real-Time Operating System

RTX51 is a multitasking real-time operating system for the 8051 family. RTX51 simplifies system and software design of complex and time-critical projects. RTX51 is a powerful tool to manage several jobs (tasks) on a single CPU. There are two distinct versions of **RTX51**:

RTX51 Full which performs both round-robin and preemptive task switching with 4 task priorities and can be operated with interrupt functions in parallel. RTX51 supports signal passing; message passing with a mailbox system and semaphores. The os_wait function of RTX51 can wait for the following events: interrupt; timeout; signal from task or interrupt; message from task or interrupt; semaphore.

RTX51 Tiny which is a subset of RTX51 Full. RTX51 Tiny easily runs on single-chip systems without off-chip memory. However, program using RTX51 Tiny can access off-chip memory. RTX51 Tiny allows round-robin task switching, supports signal passing and can be operated with interrupt functions in parallel. The os_wait function of RTX51 Tiny can wait for the following events: timeout; interval; signal from task or interrupt.

The rest of this section uses RTX-51 to refer to RTX-51 Full and RTX-51 Tiny. Differences between the two are stated where applicable.

Introduction

Many microcontroller applications require simultaneous execution of multiple jobs or tasks. For such applications, a real-time operating system (RTOS) allows flexible scheduling of system resources (CPU, memory, etc.) to several tasks. RTX-51 implements a powerful RTOS that is easy to use. RTX-51 works with all 8051 derivatives.

You write and compile RTX-51 programs using standard C constructs and compiling them with C51. Only a few deviations from standard C are required in order to specify the task ID and priority. RTX-51 programs also require that you include the **RTX51.H** or **RTX51TNY.H** header file. When you select in the µVision2 dialog Options for Target - Target the operating system, the linker adds the appropriate RTX-51 library file.

Single Task Program

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

```
int counter;

void main (void) {
  counter = 0;

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

Round-Robin Task Switching

RTX51 Tiny allows a quasi-parallel, simultaneous execution of several tasks. Each task is executed for a predefined timeout 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.

Simple C Program using RTX51

```
#include <rtx51tny.h>                      /* Definitions for RTX51 Tiny */
int counter0;
int counter1;

job0 () _task_ 0  {

  os_create_task (1);                    /* Mark task 1 as "ready"       */

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

job1 () _task_ 1  {
  while (1)  {                           /* Endless loop                 */
    counter1++;                          /* Increment counter 1          */
  }
}
```

RTX51 starts the program with task 0 (assigned to job0). The function os_create_task marks task 1 (assigned to job1) as ready for execution. These two functions are simple count loops. After the timeout period has been completed, RTX51 interrupts job0 and begins execution of job1. This function even reaches the timeout and the system continues with job0.

The os_wait Function

The os_wait function provides a more efficient way to allocate the available processor time to several tasks. os_wait interrupts the execution of the current task and waits for the specified event. During the time in which a task waits for an event, other tasks can be executed.

Wait for Timeout

RTX51 uses an 166/167 timer in order to generate cyclic interrupts (timer ticks). The simplest event argument for os_wait is a timeout, where the currently executing task is interrupted for the specified number of timer ticks. The following uses timeouts for the time delay.

Program with os_wait Function

```
#include <rtx166t.h>          /* Definitions for RTX166 Tiny */

int counter0;
int counter1;

job0 () _task_ 0  {

  os_create_task (1);

  while (1)  {
    counter0++;                /* Increment counter 0         */
    os_wait (K_TMO, 3, 0);     /* Wait 3 timer ticks          */
  }
}

job1 () _task_ 1  {
  while (1)  {
    counter1++;                /* Increment counter 1         */
    os_wait (K_TMO, 5, 0);     /* Wait 5 timer ticks          */
  }
}
```

This program is similar to the previous example with the exception that job0 is interrupted with os_wait after counter0 has been incremented. RTX166 waits three timer ticks until job0 is ready for execution again. During this time, job1 is executed. This function also calls os_wait with a timeout of 5 ticks. The result: counter0 is incremented every three ticks and counter1 is incremented every five timer ticks.

8

Wait for Signal

Another event for os_wait is a signal. Signals are used for task coordination: if a task waits with os_wait until another task issues a signal. If a signal was previously sent, the task is immediately continued.

Program with Wait for Signal.

```
#include <rtx166t.h>          /* Definitions for RTX166 Tiny */

int counter0;
int counter1;

job0 () _task_ 0  {

  os_create_task (1);

  while (1)  {
    if (++counter0 == 0) {     /* On counter 0 overflow       */
      os_send_signal (1);      /* Send signal to task 1       */
    }
  }
}

job1 () _task_ 1  {
  while (1)  {
    os_wait (K_SIG, 0, 0);     /* Wait for signal; no timeout */
    counter1++;                /* Increment counter 1         */
  }
}
```

In this example, task 1 waits for a signal from task 0 and therefore processes the overflow from counter0.

Preemptive Task Switching

The full version of RTX166 provides preemptive task switching. This feature is not included in RTX166 Tiny. It is explained here to provide a complete overview of multitasking concepts.

In the previous example, task 1 is not immediately started after a signal has arrived, but only after a timeout occurs for task 0. If task 1 is defined with a higher priority than task 0, by means of preemptive task switching, task 1 is started immediately after the signal has arrived. The priority is specified in the task definition (priority 0 is the default value).

RTX51 Technical Data

Description	RTX-51 Full	RTX-51 Tiny
Number of tasks	256; max. 19 tasks active	16
RAM requirements	40 .. 46 bytes DATA 20 .. 200 bytes IDATA (user stack) min. 650 bytes XDATA	7 bytes DATA 3 * <task count> IDATA
Code requirements	6KB .. 8KB	900 bytes
Hardware requirements	timer 0 or timer 1	timer 0
System clock	1000 .. 40000 cycles	1000 .. 65535 cycles
Interrupt latency	< 50 cycles	< 20 cycles
Context switch time	70 .. 100 cycles (fast task) 180 .. 700 cycles (standard task) depends on stack load	100 .. 700 cycles depends on stack load
Mailbox system	8 mailboxes with 8 integer entries each	not available
Memory pool system	up to 16 memory pools	not available
Semaphores	8 * 1 bit	not available

Overview of RTX51 Routines

The following table lists some of the RTX-51 functions along with a brief description and execution timing (for RTX-51 Full).

Function	Description	CPU Cycles
isr_recv_message †	Receive a message (call from interrupt).	71 (with message)
isr_send_message †	Send a message (call from interrupt).	53
isr_send_signal	Send a signal to a task (call from interrupt).	46
os_attach_interrupt †	Assign task to interrupt source.	119
os_clear_signal	Delete a previously sent signal.	57
os_create_task	Move a task to execution queue.	302
os_create_pool †	Define a memory pool.	644 (size 20 * 10 bytes)
os_delete_task	Remove a task from execution queue.	172
os_detach_interrupt †	Remove interrupt assignment.	96
os_disable_isr †	Disable 8051 hardware interrupts.	81
os_enable_isr †	Enable 8051 hardware interrupts.	80
os_free_block †	Return a block to a memory pool.	160
os_get_block †	Get a block from a memory pool.	148
os_send_message †	Send a message (call from task).	443 with task switch
os_send_signal	Send a signal to a task (call from tasks).	408 with task switch 316 with fast task switch 71 without task switch

8

Function	Description	CPU Cycles
os_send_token †	Set a semaphore (call from task).	343 with fast task switch 94 without task switch
os_set_slice †	Set the RTX-51 system clock time slice.	67
os_wait	Wait for an event.	68 for pending signal 160 for pending message

† These functions are available only in RTX-51 Full.

Additional debug and support functions in RTX-51 Full include the following:

Function	Description
oi_reset_int_mask	Disables interrupt sources external to RTX-51.
oi_set_int_mask	Enables interrupt sources external to RTX-51.
os_check_mailbox	Returns information about the state of a specific mailbox.
os_check_mailboxes	Returns information about the state of all mailboxes in the system.
os_check_pool	Returns information about the blocks in a memory pool.
os_check_semaphore	Returns information about the state of a specific semaphore.
os_check_semaphores	Returns information about the state of all semaphores in the system.
os_check_task	Returns information about a specific task.
os_check_tasks	Returns information about all tasks in the system.

CAN Functions

The CAN functions are available only with RTX-51 Full. CAN controllers supported include the Philips 82C200 and 80C592 and the Intel 82526. More CAN controllers are in preparation.

CAN Function	Description
can_bind_obj	Bind an object to a task; task is started when object is received.
can_def_obj	Define communication objects.
can_get_status	Get CAN controller status.
can_hw_init	Initialize CAN controller hardware.
can_read	Directly read an object's data.
can_receive	Receive all unbound objects.
can_request	Send a remote frame for the specified object.
can_send	Send an object over the CAN bus.
can_start	Start CAN communications.
can_stop	Stop CAN communications.
can_task_create	Create the CAN communication task.
can_unbind_obj	Disconnect the binding between a task and an object.
can_wait	Wait for reception of a bound object.
can_write	Write new data to an object without sending it.

TRAFFIC: RTX-51 Tiny Example Program

The TRAFFIC example is a pedestrian traffic light controller that shows the usage of multitasking RTX-51 Tiny Real-time operating system. During a user-defined time interval, the traffic light is operating. Outside this time interval, the yellow light flashes. If a pedestrian pushes the request button, the traffic light goes immediately into ***walk*** state. Otherwise, the traffic light works continuously.

Traffic Light Controller Commands

The serial commands that TRAFFIC supports are listed in the following table. These commands are composed of ASCII text characters. All commands must be terminated with a carriage return.

Command	Serial Text	Description
Display	D	Display clock, start, and ending times.
Time	T *hh:mm:ss*	Set the current time in 24-hour format.

Start	S *hh*:*mm*:*ss*	Set the starting time in 24-hour format. The traffic light controller operates normally between the start and end times. Outside these times, the yellow light flashes.
End	E *hh*:*mm*:*ss*	Set the ending time in 24-hour format.

Software

The TRAFFIC application is composed of three files that can be found in the **\KEIL\C51\EXAMPLES\TRAFFIC** folder.

TRAFFIC.C contains the traffic light controller program that is divided into the following tasks:

- **Task 0 init**: initializes the serial interface and starts all other tasks. Task 0 deletes itself since initialization is needed only once.
- **Task 1 command**: is the command processor for the traffic light controller. This task controls and processes serial commands received.
- **Task 2 clock**: controls the time clock.
- **Task 3 blinking**: flashes the yellow light when the clock time is outside the active time range.
- **Task 4 lights**: controls the traffic light phases while the clock time is in the active time range (between the start and end times).
- **Task 5 keyread**: reads the pedestrian push button and sends a signal to the task lights.
- **Task 6 get_escape**: If an ESC character is encountered in the serial stream the command task gets a signal to terminate the display command.

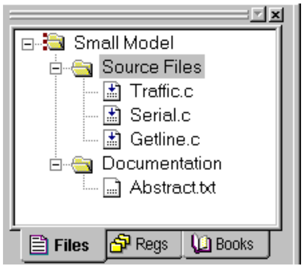
SERIAL.C implements an interrupt driven serial interface. This file contains the functions *putchar* and *getkey*. The high-level I/O functions *printf* and *getline* call these basic I/O routines. The traffic light application will also operate without using interrupt driven serial I/O. but will not perform as well.

GETLINE.C is the command line editor for characters received from the serial port. This source file is also used by the MEASURE application.

8

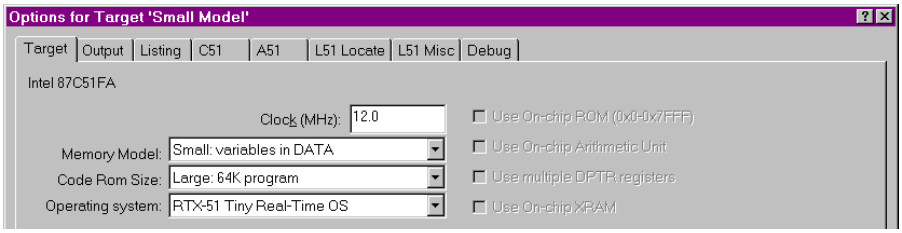
TRAFFIC Project

Open the **TRAFFIC.UV2** project file that is located in **\KEIL\C51\EXAMPLES\TRAFFIC** folder with µVision2. The source files for the TRAFFIC project will be shown in the **Project Window – Files** page.



Small Model
Source Files
Traffic.c
Serial.c
Getline.c
Documentation
Abstract.txt

The RTX-51 Tiny Real-Time OS is selected under Options for Target.



Options for Target 'Small Model'

Intel 87C51FA

Clock (MHz): 12.0

Memory Model: Small: variables in DATA

Code Rom Size: Large: 64K program

Operating system: RTX-51 Tiny Real-Time OS

□ Use On-chip ROM (0x0-0x7FFF)

□ Use On-chip Arithmetic Unit

□ Use multiple DPTR registers

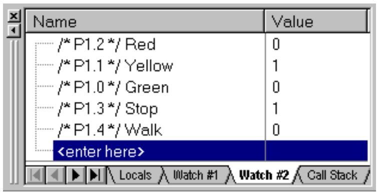
□ Use On-chip XRAM

Build the TRAFFIC program with **Project - Build** or the toolbar button.

Run the TRAFFIC Program

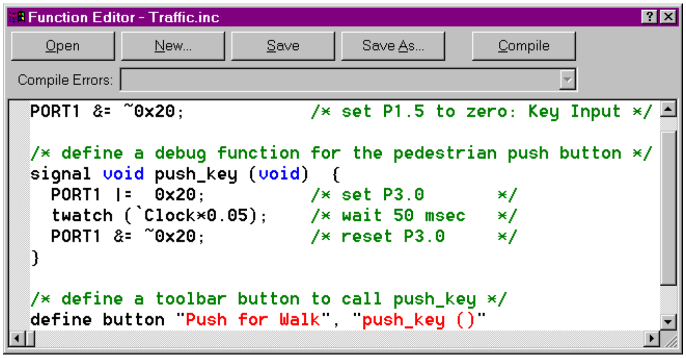
You can test TRAFFIC with the µVision2 simulator.

The watch variables shown on the right allow you to view port status that drives the lights.



Name	Value
/* P1.2 */ Red	0
/* P1.1 */ Yellow	1
/* P1.0 */ Green	0
/* P1.3 */ Stop	1
/* P1.4 */ Walk	0
<enter here>	

The **push_key** signal function simulates the pedestrian push key that switches the light system to *walk* state. This function is called with the **Push for Walk** toolbar button



Function Editor - Traffic.inc

```
PORT1 &= ~0x20;           /* set P1.5 to zero: Key Input */

/* define a debug function for the pedestrian push button */
signal void push_key (void)  {
  PORT1 |=  0x20;         /* set P3.0       */
  twatch (`Clock*0.05);   /* wait 50 msec   */
  PORT1 &= ~0x20;         /* reset P3.0     */
}

/* define a toolbar button to call push_key */
define button "Push for Walk", "push_key ()"
```

Use **Debug – Function Editor** to open **TRAFFIC.INC**. This file is specified under **Options for Target – Debug – Initialization File** and defines the signal function **push_key**, the port initialization and the toolbar button.

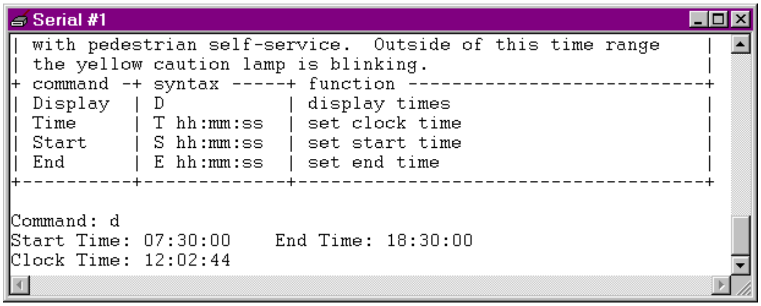
button.

Note: the VTREG symbol ***Clock*** is literalized with a back quote (**`**), since there is a C function named ***clock*** in the **TRAFFIC.C** module. Refer to “Literal Symbols” on page 109 for more information.

Now run the TRAFFIC application. Enable **View – Periodic Window Update** to view the lights in the watch window during program execution.

The **Serial Window #1** displays the *printf* output and allows you to enter the traffic light controller commands described in the table above.

Set the clock time outside of the start/end time interval to flash the yellow light.



```
| with pedestrian self-service.  Outside of this time range   |
| the yellow caution lamp is blinking.                        |
+ command -+ syntax -----+ function ---------------------------+
| Display  | D           | display times                       |
| Time     | T hh:mm:ss  | set clock time                      |
| Start    | S hh:mm:ss  | set start time                      |
| End      | E hh:mm:ss  | set end time                        |
+----------+-------------+-------------------------------------+

Command: d
Start Time: 07:30:00    End Time: 18:30:00
Clock Time: 12:02:44
```

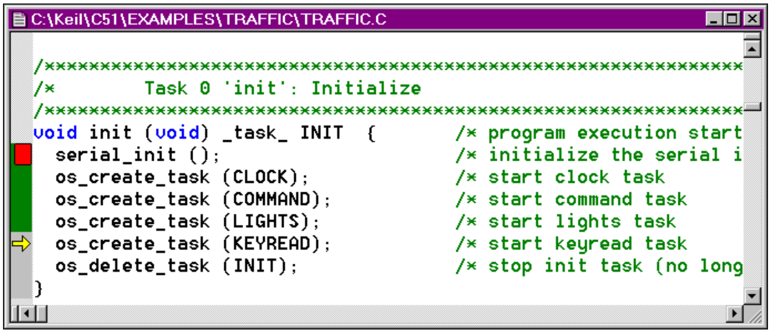
RTX Kernel Aware Debugging

A RTX application can be tested with the same methods and commands as standard 8051 applications. When you select an **Operating System** under **Options for Target – Target**, µVision2 enables additional debugging features: a dialog lists the operating system status and with the ***_TaskRunning_*** debug function you may stop program execution when a specific task is active.

The following section exemplifies RTX debugging with the TRAFFIC example.

Stop program execution, reset the CPU and kill all breakpoints.

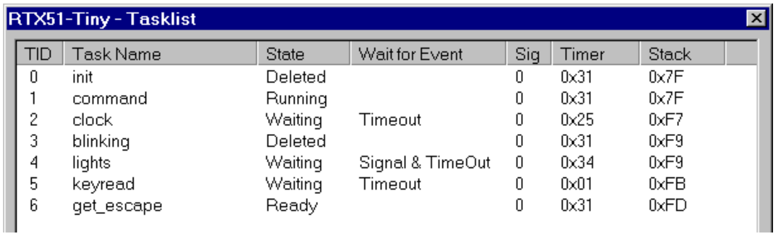
An RTX-51 application can be tested in the same way as standard applications. You may open source files, set break points and single step through the code. The TRAFFIC application starts with task 0 *init*.



C:\Keil\C51\EXAMPLES\TRAFFIC\TRAFFIC.C

```
/*****************************************************************
/*         Task 0 'init': Initialize
/*****************************************************************
void init (void) _task_ INIT  {         /* program execution start
  serial_init ();                       /* initialize the serial i
  os_create_task (CLOCK);               /* start clock task
  os_create_task (COMMAND);             /* start command task
  os_create_task (LIGHTS);              /* start lights task
  os_create_task (KEYREAD);             /* start keyread task
  os_delete_task (INIT);                /* stop init task (no long
}
```

µVision2 is completely kernel aware. You may display the task status with the menu command **Peripherals – RTX Tiny Tasklist**.



RTX51-Tiny - Tasklist

TID	Task Name	State	Wait for Event	Sig	Timer	Stack
0	init	Deleted		0	0x31	0x7F
1	command	Running		0	0x31	0x7F
2	clock	Waiting	Timeout	0	0x25	0xF7
3	blinking	Deleted		0	0x31	0xF9
4	lights	Waiting	Signal & TimeOut	0	0x34	0xF9
5	keyread	Waiting	Timeout	0	0x01	0xFB
6	get_escape	Ready		0	0x31	0xFD

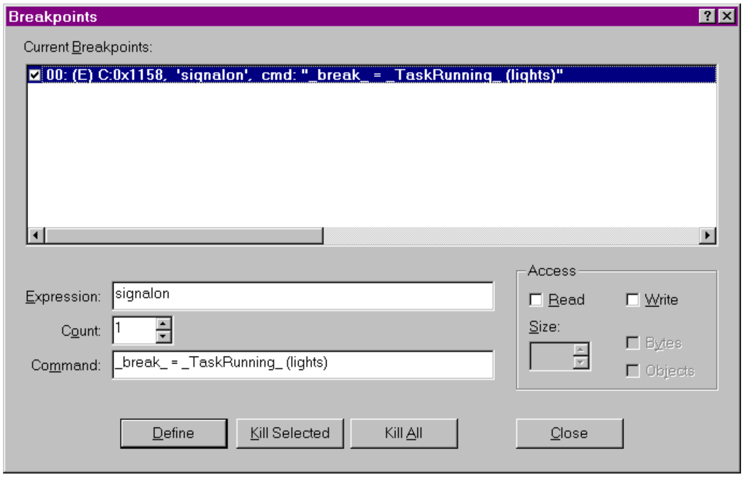
The dialog **RTX51 Tiny Tasklist** gives you the following information:

Heading	Description
TID	*task_id* used in the definition of the task function.
Task Name	name of the task function.
State	task state of the function; explained in detail in the next table.
Wait for Event	event the task is waiting for; the following events are possible (also in combination): **Timeout**: the task **Timer** is set to the duration is specified with the *os_wait* function call. After the **Timer** decrements to zero, the task goes into **Ready** state. **Interval**: the time interval specified with *os_wait* is added to the task **Timer** value. After the **Timer** decrements to zero, the task goes into **Ready** state. **Signal**: the *os_wait* function was called with **K_SIG** and the task waits for **Sig** = 1.
Sig	status of the **Signal** bit that is assigned to this task.
Timer	value of the **Timer** that is assigned to this task. The **Timer** value decrements with every RTX system timer tick. If the **Timer** becomes zero and the task is waiting for **Timeout** or **Interval** the task goes into **Ready** state.
Stack	value of the stack pointer (SP) that is used when this task is **Running**.

RTX-51 Tiny contains an efficient stack management that is explained in the "*RTX51 Tiny*" User's Guide, Chapter 5: RTX51 Tiny, Stack Management.
This manual provides detailed information about the **Stack** value.

State	Task State of a RTX166 Task Function
Deleted	Tasks that are not started are in the **Deleted** state.
Ready	Tasks that are waiting for execution are in the **Ready** state. After the currently **Running** task has finished processing, RTX starts the next task that is in the **Ready** state.
Running	The task currently being executed is in the **Running** state. Only one task is in the **Running** state at a time.
Timeout	Tasks that were interrupted by a round-robin timeout are in the **Timeout** state. This state is equivalent to **Ready**; however, a round-robin task switch is marked due to internal operating procedures.
Waiting	Tasks that are waiting for an event are in the **Waiting** state. If the event occurs which the task is waiting for, this task then enters the **Ready** state.

The **Debug – Breakpoints…** dialog allows you to define breakpoints that stop the program execution only when the task specified in the ***_TaskRunning_*** debug function argument is **Running**. Refer to "Predefined Functions" on page 122 for a detailed description of the ***_TaskRunning_*** debug function.



Breakpoints

Current Breakpoints:

☑ 00: (E) C:0x1158, 'signalon', cmd: "_break_ = _TaskRunning_ (lights)"

Expression: signalon

Count: 1

Command: _break_ = _TaskRunning_ (lights)

Access

□ Read □ Write

Size:

□ Bytes

□ Objects

The breakpoint at the function ***signalon*** stops execution only
if ***lights*** is the current **Running** task.

Chapter 9. Using on-chip Peripherals

There are a number of techniques you must know to create programs that utilize the various on-chip peripherals and features of the 8051 family. Many of these are described in this chapter. You may use the code examples provided here to quickly get started working with the 8051.

This chapter will be added in one of the next manual revisions.

Chapter 10. CPU and C Startup Code

The **STARTUP.A51** file contains the startup code for a C51 target program. This source file is located in the **LIB** directory. Include a copy of this file in each 8051 project that needs custom startup code.

This code is executed immediately upon reset of the target system and optionally performs the following operations, in order:

- Clears internal data memory
- Clears external data memory
- Clears paged external data memory
- Initializes the small model reentrant stack and pointer
- Initializes the large model reentrant stack and pointer
- Initializes the compact model reentrant stack and pointer
- Initializes the 8051 hardware stack pointer
- Transfers control to the main C function

The **STARTUP.A51** file provides you with assembly constants that you may change to control the actions taken at startup. These are defined in the following table.

Constant Name	Description
IDATALEN	Indicates the number of bytes of idata that are to be initialized to 0. The default is 80h because most 8051 derivatives contain at least 128 bytes of internal data memory. Use a value of 100h for the 8052 and other derivatives that have 256 bytes of internal data memory.
XDATASTART	Specifies the xdata address to start initializing to 0.
XDATALEN	Indicates the number of bytes of xdata to be initialized to 0. The default is 0.
PDATASTART	Specifies the pdata address to start initializing to 0.
PDATALEN	Indicates the number of bytes of pdata to be initialized to 0. The default is 0.
IBPSTACK	Indicates whether or not the small model reentrant stack pointer (**?C_IBP**) should be initialized. A value of 1 causes this pointer to be initialized. A value of 0 prevents initialization of this pointer. The default is 0.

Constant Name	Description
IBPSTACKTOP	Specifies the top start address of the small model reentrant stack area. The default is 0xFF in idata memory. C51 does not check to see if the stack area available satisfies the requirements of the applications. It is your responsibility to perform such a test.
XBPSTACK	Indicates whether or not the large model reentrant stack pointer (**?C_XBP**) should be initialized. A value of 1 causes this pointer to be initialized. A value of 0 prevents initialization of this pointer. The default is 0.
XBPSTACKTOP	Specifies the top start address of the large model reentrant stack area. The default is 0xFFFF in xdata memory. C51 does not check to see if the available stack area satisfies the requirements of the applications. It is your responsibility to perform such a test.
PBPSTACK	Indicates whether the compact model reentrant stack pointer (**?C_PBP**) should be initialized. A value of 1 causes this pointer to be initialized. A value of 0 prevents initialization of this pointer. The default is 0.
PBPSTACKTOP	Specifies the top start address of the compact model reentrant stack area. The default is 0xFF in pdata memory. C51 does not check to see if the available stack area satisfies the requirements of the applications. It is your responsibility to perform such a test.
PPAGEENABLE	Enables (a value of 1) or disables (a value of 0) the initialization of port 2 of the 8051 device. The default is 0. The addressing of port 2 allows the mapping of 256 byte variable memory in any arbitrary xdata page.
PPAGE	Specifies the value to write to Port 2 of the 8051 for pdata memory access. This value represents the xdata memory page to use for pdata. This is the upper 8 bits of the absolute address range to use for pdata. For example, if the pdata area begins at address 1000h (page 10h) in the xdata memory, **PPAGEENABLE** should be set to 1, and **PPAGE** should be set to 10h. The BL51 Linker/Locator must contain a value between 1000h and 10FFh in the PDATA control directive. For example: `BL51 <input modules> PDATA (1050H)` Neither BL51 nor C51 checks to see if the **PDATA** control directive and the **PPAGE** assembler constant are correctly specified. You must ensure that these parameters contain suitable values.

Chapter 11. Using Monitor-51

This chapter will be added in one of the next manual revisions.

Chapter 12. Command Reference

This chapter briefly describes the commands and directives for the Keil 8051 development tools. Commands and directives are listed in a tabular format along with a description.

NOTE
Underlined characters denote the abbreviation for the particular command or directive.

µVision 2 Command Line Invocation

The µVision2 IDE can directly execute operations on a project when it is called from a command line. The command line syntax is as follows:

```
UV2 ⟦command⟧ ⟦projectfile⟧
```

command is one of the following commands. If no command is specified µVision2 opens the project file in interactive **Build Mode**.

projectfile is the name of a project file. µVision2 project files have the extension .UV2. If no project file is specify, µVision2 opens the last project file used.

Command	Description
-b	Build the project and exit after the build process is complete.
-d	Start µVision2 Debugging Mode. You can use this command together with a **Debug Initialization File** to execute automated test procedures. µVision2 will exit after debugging is completed with the **EXIT** command or stop debug session. Example: `UV2 -d PROJECT1.UV2`
-r	Re-translate the project and exit after the build process is complete.
-t *targetname*	Open the project and set the specified target as current target. This option can be used in combination with other µVision2 commands. Example: `UV2 -b PROJECT1.UV2 –t"C167CR Board"` builds the target "C167CR Board" as defined in the PROJECT1.UV2 file. If the **–t** command option is not given µVision2 uses the **target** which was set as current target in the last project session.
-o *outputfile*	copy output of the **Output Window – Build** page to the specified file. Example: `UV2 -r PROJECT1.UV2 –o"listmake.prn"`

A51 / A251 Macro Assembler Directives

Invocation:

```
A51 sourcefile ⟦directives⟧
A251 @commandfile
```

`sourcefile` is the name of an assembler source file.

`commandfile` is the name of a file which contains a complete command line for the assembler including a `sourcefile` and `directives`.

`directives` are control parameters described in the following table.

A51 / A251 Controls	Meaning
CASE ‡	Enables case sensitive symbol names.
DATE(*date*)	Places ***date*** string in header (9 characters maximum).
DEBUG	Includes debugging symbol information in the object file.
ERRORPRINT⟦(*filename*)⟧	Outputs error messages to ***filename***.
INCLUDE(*filename*)	Includes the contents of ***filename*** in the assembly.
MACRO	Enables standard macro processing.
MODBIN ‡	Selects 251 binary mode (default).
MODSRC ‡	Selects 251 source mode.
MPL	Enables Intel-style macro processing.
NOAMAKE	Excludes AutoMAKE information from the object file.
NOCOND	Excludes unassembled conditional assembly code from the listing file.
NOGEN	Disables macro expansions in the listing file.
NOLINES	Excludes line number information from the object file.
NOLIST	Excludes the assembler source code from the listing file.
NOMACRO	Disables standard macro processing.
NOMOD251 ‡	Disables enhanced 251 instruction set.
NOMOD51 †	Disables predefined 8051-specific special function registers.
NOSYMBOLS	Excludes the symbol table from the listing file.
NOSYMLIST	Excludes symbol definitions from the listing file.
OBJECT⟦(*filename*)⟧, NOOBJECT	Enables or disables object file output. The object file is saved as ***filename*** if specified.
PAGELENGTH(*n*)	Sets maximum number of lines in each page of listing file.
PAGEWIDTH(*n*)	Sets maximum number of characters in each line of listing file.
PRINT⟦(*filename*)⟧, NOPRINT	Enables or disables listing file output. The listing file is saved as ***filename*** if specified.
REGISTERBANK(*num, …*), **NOREGISTERBANK**	Indicates that one or more registerbanks are used or indicates that no register banks are used.
RESET (*symbol, …*)	Assigns a value of 0000h to the specified symbols.
SET (*symbol, …*)	Assigns a value of 0FFFFh to the specified symbols.

A51 / A251 Controls	Meaning
TITLE(*title*)	Includes ***title*** in the listing file header.
XREF	Includes a symbol cross reference listing in the listing file.

† These controls are available only in the A51 macro assembler.
‡ These controls are available only in A251 macro assembler.

C51/C251 Compiler

Invocation:

```
C51 sourcefile [directives]

C251 sourcefile [directives]

C51 @commandfile

C251 @commandfile
```

where

`sourcefile` is the name of a C source file.

`commandfile` is the name of a file which contains a complete command line for the compiler including a **`sourcefile`** and **`directives`**. You may use a command file to make compiling a source file easier or when you have more directives than fit on the command line.

`directives` are control parameters which are described in the following table.

C51 / C251 Controls	Meaning
CODE	Includes an assembly listing in the listing file.
COMPACT	Selects the COMPACT memory model.
DEBUG	Includes debugging information in the object file.
DEFINE	Defines preprocessor names on the command line.
FLOATFUZZY	Specifies the number of bits rounded during floating-point comparisons.
HOLD(*d*,*n*,*x*) ‡	Specifies size limits for variables placed in **data** (***d***), **near** (***n***), and **xdata** (***x***) memory areas.
INTERVAL †	Specifies the interval for interrupt vectors.
INTR2 ‡	Saves upper program counter byte and PSW1 in interrupt functions.
INTVECTOR(*n*), NOINTVECTOR	Specifies offset for interrupt table, using ***n***, or excludes interrupt vectors from the object file.
LARGE	Selects the LARGE memory model.
LISTINCLUDE	Includes the contents of include files in the listing file.
MAXARGS(*n*)	Specifies the number of bytes reserved for variable length argument lists.

C51 / C251 Controls	Meaning
MOD517 †	Enables support for the additional hardware of the Siemens 80C517 and its derivatives.
MODBIN ‡	Generates 251 binary mode code.
MODDP2 †	Enables support for the additional hardware of Dallas Semiconductor 80C320/520/530 and the AMD 80C521.
MODSRC ‡	Generates 251 source mode code.
NOAMAKE	Excludes AutoMAKE information from the object file.
NOAREGS †	Disables absolute register addressing using **AR*n*** instructions.
NOCOND	Excludes skipped conditional code from the listing file.
NOEXTEND	Disables 8051/251 extensions and processes only ANSI C constructs.
NOINTPROMOTE †	Disables ANSI integer promotion rules.
NOREGPARMS †	Disables passing parameters in registers.
OBJECT⟦**(*filename*)**⟧**, NOOBJECT**	Enables or disables object file output. The object file is saved as ***filename*** if specified.
OBJECTEXTEND †	Includes additional variable type information in the object file.
OPTIMIZE	Specifies the level of optimization performed by the compiler.
ORDER	Locates variables in memory in the same order in which they are declared in the source file.
PAGELENGTH(*n*)	Sets maximum number of lines in each page of listing file.
PAGEWIDTH(*n*)	Sets maximum number of characters in each line of listing file.
PARM51 ‡	Uses parameter passing conventions of the C51 compiler.
PREPRINT⟦**(*filename*)**⟧	Produces a preprocessor listing file with all macros expanded. The preprocessor listing file is saved as ***filename*** if specified.
PRINT⟦**(*filename*)**⟧**, NOPRINT**	Enables or disables listing file output. The listing file is saved as ***filename*** if specified.
REGFILE(*filename*)	Specifies the name of the generated file to contain register usage information.
REGISTERBANK †	Selects the register bank to use functions in the source file.
ROM({SMALL|COMPACT|LARGE})	Controls generation of **AJMP** and **ACALL** instructions.
SMALL	Selects the SMALL memory model.
SRC	Creates an assembly source file instead of an object file.
SYMBOLS	Includes a list of the symbols used in the listing file.
WARNINGLEVEL(*n*)	Controls the types and severity of warnings generated.

† These controls are available only in the C51 compiler.

‡ These controls are available only in C251 compiler.

L51/BL51 Linker/Locator

Invocation:

```
BL51 inputlist [TO outputfile] [directives]

L51 inputlist [TO outputfile] [directives]

BL51 @commandfile

L51 @commandfile
```

where

`inputlist` is a list of the object files and libraries, separated by commas, that the linker includes in the final 8051 application.

`outputfile` is the name of the absolute object module the linker creates.

`commandfile` is the name of a file which contains a complete command line for the linker/locator including an **`inputlist`** and **`directives`**. You may use a command file to make linking your application easier or when you have more input files or more directives than fit on the command line.

`directives` are control parameters which are described in the following table.

BL51 Controls	Meaning
BANKAREA ‡	Specifies the address range where the code banks are located.
BANK*x* ‡	Specifies the starting address, segments, and object modules for code banks 0 to 31.
BIT	Locates and orders **BIT** segments.
CODE	Locates and orders **CODE** segments.
COMMON ‡	Specifies the starting address, segments, and object modules to place in the common bank. This directive is essentially the same as the **CODE** directive.
DATA	Locates and orders **DATA** segments.
IDATA	Locates and orders **IDATA** segments.
IXREF	Includes a cross reference report in the listing file.
NAME	Specifies a module name for the object file.
NOAMAKE	Excludes AutoMAKE information from the object file.
NODEBUGLINES	Excludes line number information from the object file.
NODEBUGPUBLICS	Excludes public symbol information from the object file.
NODEBUGSYMBOLS	Excludes local symbol information from the object file.
NODEFAULTLIBRARY	Excludes modules from the run-time libraries.
NOLINES	Excludes line number information from the listing file.
NOMAP	Excludes memory map information from the listing file.

BL51 Controls	Meaning
NOOVERLAY	Prevents overlaying or overlapping local **BIT** and **DATA** segments.
NOPUBLICS	Excludes public symbol information from the listing file.
NOSYMBOLS	Excludes local symbol information from the listing file.
OVERLAY	Directs the linker to overlay local data & bit segments and lets you change references between segments.
PAGELENGTH(*n*)	Sets maximum number of lines in each page of listing file.
PAGEWIDTH(*n*)	Sets maximum number of characters in each line of listing file.
PDATA	Specifies the starting address for **PDATA** segments.
PRECEDE	Locates and orders segments that should precede all others in the internal data memory.
PRINT	Specifies the name of the listing file.
RAMSIZE	Specifies the size of the on-chip data memory.
REGFILE(*filename*)	Specifies the name of the generated file to contain register usage information.
RTX51 ‡	Includes support for the RTX-51 full real-time kernel.
RTX51TINY ‡	Includes support for the RTX-51 tiny real-time kernel.
STACK	Locates and orders **STACK** segments.
XDATA	Locates and orders **XDATA** segments.

‡ These controls are available only in the BL51 code banking linker/locator.

L251 Linker/Locator

Invocation:

```
L251 inputlist [TO outputfile] [directives]

L251 @commandfile
```

where

`inputlist` is a list of the object files and libraries, separated by commas, that the linker includes in the final 251 application.

`outputfile` is the name of the absolute object module the linker creates.

`commandfile` is the name of a file which contains a complete command line for the linker/locator including an `inputlist` and `directives`. You may use a command file to make linking your application easier or when you have more input files or more directives than fit on the command line.

`directives` are control parameters which are described in the following table.

L251 Controls	Meaning
ASSIGN	Defines public symbols on the command line.
CLASSES	Specifies a physical address range for segments in a memory class.
IXREF	Includes a cross reference report in the listing file.
NAME	Specifies a module name for the object file.
NOAMAKE	Excludes AutoMAKE information from the object file.
NOCOMMENTS	Excludes comment information from the listing file and the object file.
NODEFAULTLIBRARY	Excludes modules from the run-time libraries.
NOLINES	Excludes line number information from the listing file and object file.
NOMAP	Excludes memory map information from the listing file.
NOOVERLAY	Prevents overlaying or overlapping local **BIT** and **DATA** segments.
NOPUBLICS	Excludes public symbol information from the listing file and the object file.
NOSYMBOLS	Excludes local symbol information from the listing file.
NOTYPES	Excludes type information from the listing file and the object file.
OBJECTCONTROLS	Excludes specific debugging information from the object file. Subcontrols must be specified in parentheses. See **NOCOMMENTS**, **NOLINES**, **NOPUBLICS**, **NOSYMBOLS**, and **PURGE**.
OVERLAY	Directs the linker to overlay local data & bit segments and lets you change references between segments.
PAGELENGTH(*n*)	Sets maximum number of lines in each page of listing file.
PAGEWIDTH(*n*)	Sets maximum number of characters in each line of listing file.
PRINT	Specifies the name of the listing file.
PRINTCONTROLS	Excludes specific debugging information from the listing file. Subcontrols must be specified in parentheses. See **NOCOMMENTS**, **NOLINES**, **NOPUBLICS**, **NOSYMBOLS**, and **PURGE**.
PURGE	Excludes all debugging information from the listing file and the object file.
RAMSIZE	Specifies the size of the on-chip data memory.
REGFILE(*filename*)	Specifies the name of the generated file to contain register usage information.
RESERVE	Reserves memory ranges and prevents the linker from using these memory areas.
RTX251	Includes support for the RTX-251 full real-time kernel.
RTX251TINY	Includes support for the RTX-251 tiny real-time kernel.
SEGMENTS	Defines physical memory addresses and orders for specified segments.
SEGSIZE	Specifies memory space used by a segment.

L251 Controls	Meaning
<u>W</u>ARNING<u>L</u>EVEL(*n*)	Controls the types and severity of warnings generated.

LIB51 / L251 Library Manager Commands

The LIB51 / LIB251 Library Manager lets you create and maintain library files of your 8051 / 251 object modules. Invoke the library manager using the following command:

```
LIB51  ⟦command⟧
LIB251 @commandfile
```

command is one of the following commands. If no command is specified LIB51 / LIB251 enters an interactive command mode.

commandfile is the name of a file which contains a complete command line for the library manager. The command file includes a single ***command*** that is executed by LIB51. You may use a command file to generate a large library with at once.

LIB51 Command	Description
ADD	Adds an object module to the library file. For example, `LIB51 ADD GOODCODE.OBJ TO MYLIB.LIB` adds the **GOODCODE.OBJ** object module to **MYLIB.LIB**.
CREATE	Creates a new library file. For example, `LIB251 CREATE MYLIB.LIB` creates a new library file named **MYLIB.LIB**.
DELETE	Removes an object module from the library file. For example, `LIB51 DELETE MYLIB.LIB (GOODCODE)` removes the **GOODCODE** module from **MYLIB.LIB**.
EXTRACT	Extracts an object module from the library file. For example, `LIB251 EXTRACT MYLIB.LIB (GOODCODE) TO GOOD.OBJ` copies the **GOODCODE** module to the object file **GOOD.OBJ**.
EXIT	Exits the library manager interactive mode.
HELP	Displays help information for the library manager.
LIST	Lists the module and public symbol information stored in the library file. For example, `LIB251 LIST MYLIB.LIB TO MYLIB.LST PUBLICS` generates a listing file (named **MYLIB.LST**) that contains the module names stored in the **MYLIB.LIB** library file. The `PUBLICS` directive specifies that public symbols are also included in the listing.
REPLACE	Replaces an existing object module to the library file. For example, `LIB51 REPLACE GOODCODE.OBJ IN MYLIB.LIB` replaces the **GOODCODE.OBJ** object module in **MYLIB.LIB**. Note that Replace will add **GOODCODE.OBJ** to the library if it does not exists.
TRANSFER	Generates a complete new library and adds object modules. For example, `LIB251 TRANSFER FILE1.OBJ, FILE2.OBJ TO MYLIB.LIB` deletes the existing library **MYLIB.LIB**, re-creates it and adds the object modules **FILE1.OBJ and FILE2.OBJ** to that library.

OC51 Banked Object File Converter

Invocation:

```
OC51 banked_file
```

where

`banked_file` is the name of a banked object file.

OH51 Object-Hex Converter

Invocation:

```
OH51 absfile [HEXFILE(hexfile)]
```

where

`absfile` is the name of an absolute object file.

`hexfile` is the name of the Intel HEX file to create.

OH251 Object-Hex Converter

Invocation:

```
OH251 absfile [HEXFILE(hexfile)] [{HEX|H386}]
              [RANGE(start-end)]
```

where

`absfile` is the name of an absolute object file.

`hexfile` is the name of the HEX file to create.

`HEX` specifies that a standard Intel HEX file is created.

`H386` specifies that an Intel HEX-386 file is created.

`RANGE` specifies the address range of data in the **`absfile`** to convert and store in the HEX file. The default range is 0xFF0000 to 0xFFFFFF.

`start` specifies the starting address of the range. This address must be entered in C hexadecimal notation, for example: **`0xFF0000`**.

`end` specifies the ending address of the range. This address must be entered in C hexadecimal notation, for example: **`0xFFFFFF`**.

12

Index