



# **Getting Started and Create Applications**

**with  $\mu$ Vision2 and 166/ST10  
Microcontroller Development Tools**

**User's Guide 03.99**

Information in this document is subject to change without notice and does not represent a commitment on the part of the manufacturer. The software described in this document is furnished under license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than for the purchaser's personal use, without written permission.

Copyright © 1997-1999 Keil Elektronik GmbH and Keil Software, Inc.  
All rights reserved.

Keil C166™ and  $\mu$  Vision™ are trademarks of Keil Elektronik GmbH.  
Microsoft®, and Windows™ are trademarks or registered trademarks of Microsoft Corporation.  
PC® is a registered trademark of International Business Machines Corporation.

---

**NOTE**

*This manual assumes that you are familiar with Microsoft Windows and the hardware and instruction set of the 166 and ST10 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 Siemens 166 and ST Microelectronics ST10 family of microcontrollers. It introduces new users and interested readers to our product line. With nothing more than this book, you should be able to successfully run and use our tools. This user's guide contains the following chapters.

“Chapter 1. Introduction” gives an overview of this user's guide and discusses the different products that we offer for the 166 and ST10 microcontroller family.

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

“Chapter 3. Development Tools” describes the major features of the  $\mu$ Vision2 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  $\mu$ Vision2 debugger to simulate and test your entire application.

“Chapter 6.  $\mu$ Vision2 Debug Functions” discusses built-in, user, and signal functions that extended the debug capabilities or generate I/O signals.

“Chapter 7. Sample Programs” shows how to use the Keil 166 development tools.

“Chapter 8. Using on-chip Peripherals” shows how to access the on-chip 166/ST10 peripherals within the C166 compiler.

“Chapter 10. CPU and C Startup Code” provides information on setting up the 166/ST10 CPU for your application.

“Chapter 11. Using Monitor-166” 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 166 development tools.

# Document Conventions

This document uses the following conventions:

Examples	Description
<b>README.TXT</b>	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:  <div style="text-align: center;"> <b>CLS                    DIR                    L166.EXE</b> </div>
	Note that you are not required to enter these commands using all capital letters.
<b>Courier</b>	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.
<i>Variables</i>	Text in italics represents information that you must provide. For example, <i>projectfile</i> in a syntax string means that you must supply the actual project file name.  Occasionally, italics are also used to emphasize words in the text.
Elements that repeat...	Ellipses (...) are used to indicate an item that may be repeated.
Omitted code : : :	Vertical ellipses are used in source code listings to indicate that a fragment of the program is omitted. For example:  <pre> <b>Void main (void) {</b> <b>  :</b> <b>  :</b> <b>  while (1);</b> </pre>
[ <i>Optional Items</i> ]	Optional arguments in command lines and optional fields are indicated by double brackets. For example:  <b>C166 TEST.C PRINT [ ( <i>filename</i> ) ]</b>
{ <i>opt1</i>   <i>opt2</i> }	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.
<b>Keys</b>	Text in this sans serif typeface represents actual keys on the keyboard. For example, "Press <b>Enter</b> to continue."
<b>Point</b>	Move the mouse until the mouse pointer rests on the item desired.
<b>Click</b>	Quickly press and release a mouse button while pointing at the item to be selected.
<b>Drag</b>	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.
<b>Double-Click</b>	Click the mouse button twice in rapid succession.

# Contents

<b>Chapter 1. Introduction .....</b>	<b>1</b>
166/ST10 Microcontroller Family .....	2
Manual Topics .....	3
Changes to the Documentation .....	3
Evaluation Kits and Production Kits .....	4
Types of Users .....	4
Requesting Assistance .....	5
Product Overview .....	8
<b>Chapter 2. Installation.....</b>	<b>11</b>
System Requirements .....	11
Installation Details .....	11
Folder Structure .....	12
<b>Chapter 3. Development Tools.....</b>	<b>13</b>
$\mu$ Vision2 Integrated Development Environment .....	13
C166 Optimizing C Cross Compiler .....	20
A166 Macro Assembler .....	28
L166 Linker/Locator .....	30
LIB166 Library Manager .....	34
OH166 Object-HEX Converter .....	34
<b>Chapter 4. Creating Applications .....</b>	<b>35</b>
Create a Project .....	35
Project Targets and File Groups .....	42
Overview of Configuration Dialogs .....	43
$\mu$ Vision2 Utilities .....	44
Writing Optimum Code.....	48
Applications without external RAM Devices .....	55
Tips and Tricks .....	56
<b>Chapter 5. Testing Programs.....</b>	<b>67</b>
$\mu$ Vision2 Debugger .....	67
Debug Commands .....	80
Expressions .....	82
<b>Chapter 6. <math>\mu</math>Vision2 Debug Functions .....</b>	<b>97</b>
Creating Functions .....	97
Invoking Functions .....	99
Function Classes .....	99
Differences Between Debug Functions and C .....	110
<b>Chapter 7. Sample Programs .....</b>	<b>111</b>
HELLO: Your First 166 C Program.....	112
MEASURE: A Remote Measurement System .....	117

---

<b>Chapter 8. Using on-chip Peripherals.....</b>	<b>127</b>
DPP Registers .....	129
Interrupts .....	131
Peripheral Event Controller .....	134
Parallel Port I/O.....	138
General Purpose Timers.....	140
Serial Interface .....	142
Watchdog Timer .....	145
Pulse Width Modulation .....	146
A/D Converter .....	149
Power Reduction Modes.....	150
<b>Chapter 10. CPU and C Startup Code .....</b>	<b>153</b>
Selecting the Memory Model .....	153
Configuring the Startup Code .....	153
<b>Chapter 11. Using Monitor-166.....</b>	<b>155</b>
$\mu$ Vision2 Monitor Driver .....	157
Target Options when Using Monitor-166 .....	159
Monitor-166 Configuration.....	160
Trouble Shooting .....	160
<b>Chapter 12. Command Reference.....</b>	<b>163</b>
$\mu$ Vision 2 Command Line Invocation.....	163
A166 Macro Assembler Directives .....	164
C166 Optimizing C Cross Compiler Directives .....	166
L166 Linker/Locator Directives .....	168
LIB166 Library Manager Commands.....	170
OH166 Object-HEX Converter Commands .....	171
<b>Index .....</b>	<b>172</b>

# Chapter 1. Introduction

Thank you for allowing Keil Software to provide you with software development tools for the 166 and ST10 family of microprocessors. With the Keil tools, you can generate embedded applications for the C161, C163, C164, C165, 8xC166, C167 and ST10 microcontrollers as well as future derivatives.

---

**NOTE**

*Throughout this manual we refer to these tools as the **166** development tools. However, they are not limited to just the 8xC166 devices.*

---

The Keil Software 166 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 13.

- $\mu$ Vision2 for Windows™ Integrated Development Environment: combines Project Management, Source Code Editing, and Program Debugging in one powerful environment.
- C166 ANSI Optimizing C Cross Compiler: creates relocatable object modules from your C source code,
- A166 Macro Assembler: creates relocatable object modules from your 8xC166 or C167 assembler source code,
- L166 Linker/Locator: combines relocatable object modules created by the compiler and assembler into the final absolute object module,
- LIB166 Library Manager: combines object modules into a library which may be used by the linker,
- OH166 Object-HEX Converter: creates Intel HEX files from absolute object modules,
- RTX-166 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 166 hardware.

## 166/ST10 Microcontroller Family

The 166/ST10 family of microcontrollers has been available since the early 1990's. With a wide variety of outstanding features and peripherals, the 166 CPU core is destined to see service well into the next century. Many derivatives are available and several are available from multiple sources (Siemens and ST Microelectronics). As a 16-bit, high-performance embedded processor, the 166 family has no equal.

A typical 166/ST10 family member contains the 8xC166 or C167 CPU core, data memory, code memory, and some versatile peripheral functions. A flexible memory interface lets you expand the capabilities of the 166 using standard peripherals and memory devices in either 8-bit or 16-bit configurations.

### Overview of various 166 and ST10 derivatives

The 166/ST10 family covers today 30+ different CPU variants with extensive on-chip peripheral and I/O. Many variants offer A/D converters, CAN interface, Flash memory, and a rich set of timers, interrupts and I/O pins. Also included are power management features, watchdogs and clock drivers. The memory interface allows a mixture of 8 / 16-bit multiplexed / non-multiplex bus systems including chip select pins on up to five memory areas. The following list provides only a brief overview.

Basic Device Code	max. CPU Clock	Description
C161 (several Variants)	16 MHz	low-cost controller. Variants with on-chip Flash memory and CAN interface are available.
C163	25 MHz	down-grade of the C165 CPU, less on-chip RAM.
C164	20 MHz	with special capture/compare unit for high-speed signal generation and event capturing. variants with on-chip OTP memory available.
C165	25 MHz	down-grade of the C167 CPU, no A/D converter, less I/O.
8xC166	20 MHz	The original 166 CPU; no extended instruction set and limited to 256 KB memory space. Variants with Flash memory available.
C167 (several Variants)	25 MHz	The high-end 166 CPU with extensive peripherals and I/O capabilities. Some variants have on-chip Flash program and data memory.
ST10-272	50 MHz	version with high speed CPU and on-chip MAC (DSP) co-processor unit.

This list represents the devices available in January 1999. The 166/ST10 microcontroller family constantly grows. Announcements for 1999: several new devices and improving of the CPU speed.



## 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 166 development tools (see “Chapter 3. Development Tools” on page 13),
- Create full applications using the  $\mu$ Vision2 integrated development environment (see “Chapter 4. Creating Applications” on page 35),
- Debug programs and simulate target hardware using the  $\mu$ Vision2 debugger (see “Chapter 5. Testing Programs” on page 67),
- Access the on-chip peripherals and special features of the 166 variants using the C166 compiler (see “Chapter 8. Using on-chip Peripherals” on page 127),
- Run the included sample programs (see “Chapter 7. Sample Programs” on page 111).

---

### **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 111).*

---

## 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 **C166\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 **EK166 Evaluation Kit** includes evaluation versions of our 166 tools along with this user's guide. The tools in the evaluation kit let you generate applications up to 4 Kbytes in size. You may use this kit to evaluate the effectiveness of our 166 tools and to generate small target applications.

The **166 Production Kits** (discussed in "Product Overview" on page 8) include the unlimited versions of our 166 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](http://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 4 Kbytes along with several sample programs that provide real-world applications created for the 166/ST10 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 166 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 166 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 166 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 pertain 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](http://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  $\mu$ Vision2 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](http://www.keil.com).*

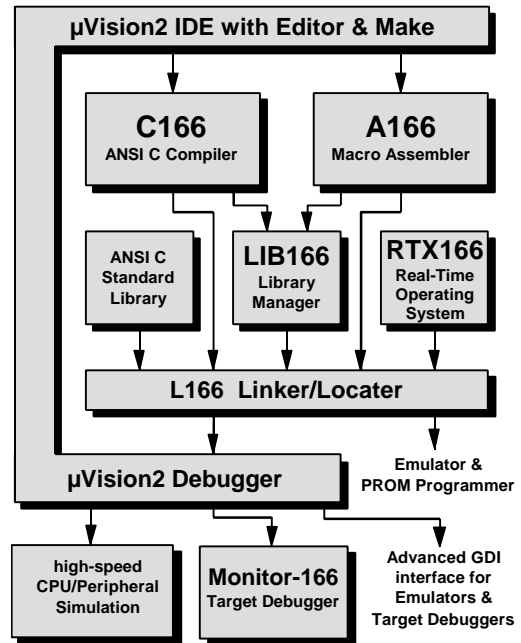
---

## 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 166/ST10 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 166 tool set.



### 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 your development efforts.

### 166 Compiler & Assembler

Source files are created by the µVision2 IDE and are passed to the C166 compiler or A166 assembler. The compiler and assembler process source files and creates relocatable object files. The Keil C166 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 166 environment have been added. The Keil A166 macro assembler supports the complete instruction sets of the 8xC166, C167 and ST10 derivatives.

## LIB166 Library Manager

Object files created by the compiler and assembler may be used by the LIB166 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.

## L166 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  $\mu$ Vision2 Debugger or with an in-circuit emulator for the program test.

## $\mu$ Vision2 Debugger

The  $\mu$ Vision2 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 166 system including on-chip peripherals and external hardware. Via the integrated device database you can configure the  $\mu$ Vision2 debugger to the attributes and peripherals of 166/ST10 device you are using.

For testing the software in a real hardware, you may connect the  $\mu$ Vision2 Debugger with Monitor-166 or you can use the *Advanced GDI* interface to attach the debugger front-end to a target system.

## Monitor-166

The  $\mu$ Vision2 Debugger supports target debugging using Monitor-166. The monitor program is a program that resides in the memory of your target hardware and communicates with  $\mu$ Vision2 using the serial port of the 166 and a COM port of your PC. With Monitor-166,  $\mu$ Vision2 lets you perform source-level, symbolic debugging on your target hardware.

## RTX166 Real-Time Operating System

The RTX166 real-time operating system is a multitasking kernel for the 166 family. The RTX166 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 C166 compiler and is easy to use. Task description tables and operating system consistency are automatically controlled by the L166 linker/locator.

## Product Overview

Keil Software provides the premier development tools for the Siemens 166 and ST Microelectronics ST10 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 166 development tools. Each kit and its contents are described below.

### PK166 Professional Developer’s Kit

The **PK166** Professional Developer’s Kit includes everything the professional developer needs to create and debug sophisticated embedded applications for the Siemens C161, C163, C164, C165, 8xC166, and C167 as well as the ST Microelectronics ST10 series of microcontrollers. The professional developer’s kit can be configured for all 166/ST10 derivatives.

### PK161 Professional Developer’s Kit

The PK161 Professional Developer’s Kit is a reduced version of PK166 and can be used only for the Siemens C161 derivatives. Other 166/ST10 family members are not supported.

### CA166 Compiler Kit

The CA166 Compiler Kit is the best choice for developers who need a C compiler but not a debugging system. The CA166 package contains only the  $\mu$ Vision IDE. The  $\mu$ Vision2 Debugger features are not available in CA166. The kit includes everything you need to create embedded applications and can be configured for all 166/ST10 derivatives.

### A166 Assembler Kit

The A166 Assembler Kit includes an assembler and all the utilities you need to create embedded applications. It can be configured for all 166/ST10 derivatives.

## RTX166 Real-Time Operating System (FR166)

The RTX166 Real-Time Operating Systems is a real-time kernel for the 166 family of microcontrollers. RTX166 Full provides a superset of the features found in RTX166 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.

Support	PK166	CA166	A166	PK161 <sup>†</sup>	FR166
µVision2 Project Management & Editor	✓	✓	✓	✓	
A166 Assembler	✓	✓	✓	✓	
C166 Compiler	✓	✓		✓	
L166 Linker/Locator	✓	✓	✓	✓	
LIB166 Library Manager	✓	✓	✓	✓	
µVision2 Debugger/Simulator	✓			✓	
RTX166 Tiny	✓	✓		✓	
RTX166 Full					✓

<sup>†</sup> PK161 supports only C161 derivatives; it does not include support for other 166/ST10 devices.

**1**



## 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.

# 2

### 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 166 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 166 development tools. Your installation may vary depending on the products you purchased.

Folder	Description
<b>C:\KEIL\C166\ASM</b>	Assembler SFR definition files and template source file.
<b>C:\KEIL\C166\BIN</b>	Executable files of the 166 toolchain.
<b>C:\KEIL\C166\CAN</b>	RTX166 Full CAN example programs.
<b>C:\KEIL\C166\EXAMPLES</b>	Sample applications.
<b>C:\KEIL\C166\RTX166</b>	RTX166 Full files.
<b>C:\KEIL\C166\RTX_TINY</b>	RTX166 Tiny files.
<b>C:\KEIL\C166\INC</b>	C compiler include files.
<b>C:\KEIL\C166\LIB</b>	C compiler library files, startup code, and source of I/O routines.
<b>C:\KEIL\C166\MONITOR</b>	Target Monitor files and Monitor configuration for user hardware.
<b>C:\KEIL\UV2</b>	Generic $\mu$ Vision2 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 166 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.

### $\mu$ Vision2 Integrated Development Environment

$\mu$ Vision2 is a standard Windows application.  $\mu$ Vision2 is an integrated software development platform that combines a robust editor, project manager, and make facility.  $\mu$ Vision2 supports all of the Keil tools for the 166 including the C compiler, macro assembler, linker/locator, and object-HEX converter.  $\mu$ Vision2 helps expedite the development process of your embedded applications by providing the following:

- e Full-featured source code editor,
- e Device Database for pre-configuring the development tool setting,
- e Project manager for creating and maintaining your projects,
- e Integrated make facility for assembling, compiling, and linking your embedded applications,
- e Dialogs for all development tool settings,
- e True integrated source-level Debugger with high-speed CPU and peripheral simulator.
- e Advanced GDI interface for software debugging in the target hardware and for connection to Monitor-166.
- e Links to development tools manuals, device datasheets & user's guides.

---

#### **NOTE**

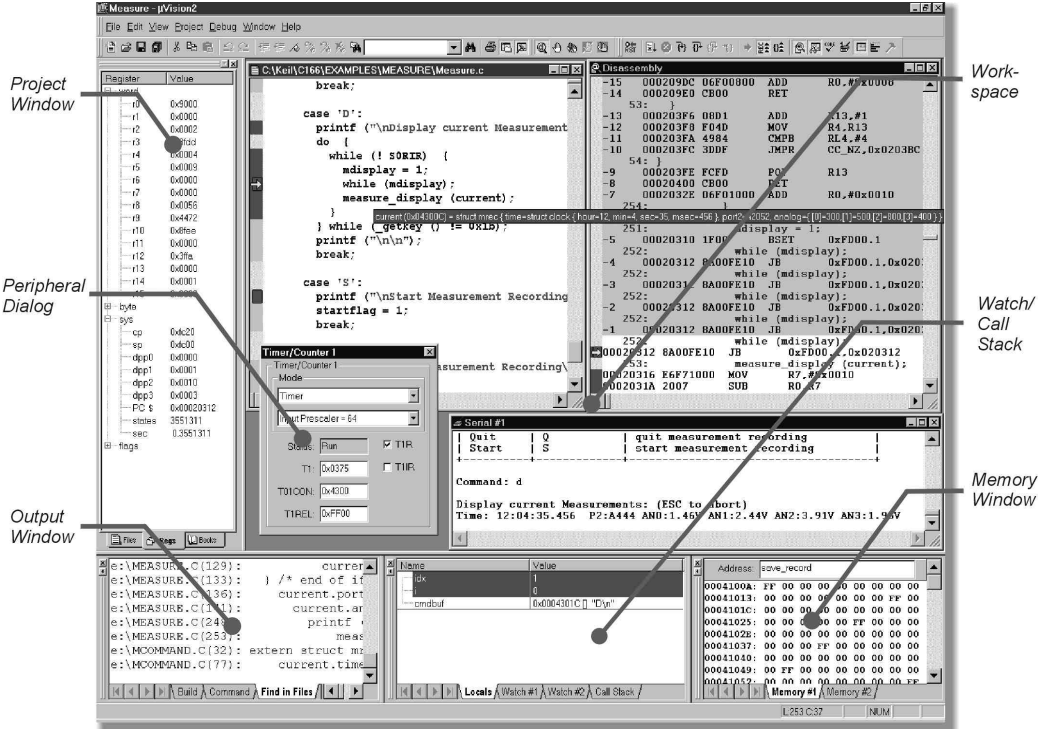
*The  $\mu$ Vision2 debugging features are only available in the **PK166** and **PK161** tool kits.*

---

## About the Environment

The  $\mu$ Vision2 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.  $\mu$ Vision2 lets you simultaneously open and view multiple source files.

3







## 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. Several commands can be reached also with keyboard shortcuts. The following tables give you an overview of the  $\mu$ Vision2 commands.


### File Menu and File Commands

Toolbar	File Menu	Shortcut	Description
	New	Ctrl+N	Create a new source or text file

Toolbar	File Menu	Shortcut	Description
	Open	Ctrl+O	Open an existing file
	Close		Close the active file
	Save	Ctrl+S	Create a new source or text file
	Save as...		Save all open source and text files
	Device Database		Maintain the $\mu$ Vision2 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 $\mu$ Vision2 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 right
		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
	Find	Ctrl+F	Search text in the active file
		F3	Repeat search text forward
		Shift+F3	Repeat search text backward
		Ctrl+F3	Search word under cursor
		Ctrl+]	Find matching brace, parenthesis, or bracket
	Find in Files...		Search text in several files
	Replace	Ctrl+H	Replace specific text
			Indent selected text right one tab stop
			Indent selected text left one tab stop
		Ctrl+F2	Toggle bookmark at current line
		F2	Move cursor to next bookmark
		Shift+F2	Move cursor to previous bookmark

Toolbar	Edit Menu	Shortcut	Description
			Clear all bookmarks in active file

## Select Text Commands















In  $\mu$ Vision2 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 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
	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...		Select Colors & Fonts and Editor options

## Project Menu and Project Commands

Toolbar	Project Menu	Shortcut	Description
	New Project ...		Create a new project
	Open Project ...		Open an existing project
	Close Project...		Close current project
	Targets, Groups, Files		Maintain Targets, File Groups and Files of a proeject
	Select Device for Target		Select a CPU from the Device Database
	Options...	Alt+F7	Change tool options for Target, Group or File
			Change options for current Target
	MCB167		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

3

## Debug Menu and Debug Commands

Toolbar	Debug Menu	Shortcut	Description
	Start/Stop Debugging	Ctrl+F5	Start or stop $\mu$ Vision2 Debug Mode
	Reset CPU		Set CPU to reset state
	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
	Enable/Disable Trace Recording		Enable trace recording for instruction review
	View Trace Records		Review previous executed instructions
	Breakpoints...		Open Breakpoint dialog
			Toggle breakpoint on current line
			Kill all breakpoints in the program
			Enable/disable breakpoint on the current line
			Disable all breakpoints in the program
			Show next executeable statement/instruction
	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

Toolbar	Debug Menu	Shortcut	Description
	[ 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 run custom programs. The menu is extended once you have added customer programs with the option Customize Tools Menu...

Toolbar	Tools Menu	Shortcut	Description
	Target Environment		Configure compiler and linker paths
	PC-Lint Options		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
	Customize Tools Menu...		Add user programs to the Tools Menu

## 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 $\mu$ Vision		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 35.
- **Debug Mode:** provides you with a powerful debugger for testing your application. The Debug Mode is described in "Chapter 5. Testing Programs" on page 67.

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

## C166 Optimizing C Cross Compiler

For 166/ST10 microcontroller applications, the Keil C166 Cross Compiler offers a way to program in C which truly matches assembly programming in terms of code efficiency and speed. The Keil C166 is not a universal C compiler adapted for the 166. It is a dedicated C compiler that generates extremely fast and compact code. The Keil C166 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 166/ST10 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.

### C166 Language Extensions

The C166 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 166 microprocessor. The C166 compiler includes extensions for:

- e Data Types,
- e Memory Types,
- e Memory Models,
- e Pointers,
- e Reentrant Functions,
- e Interrupt Functions,
- e Real-Time Operating Systems.

The following sections briefly describe these extensions.

## Data Types

The C166 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.

Data Type	Size	Range of values
bit	1 bit	0 or 1
signed char	1 byte	-128 to +127
unsigned char	1 byte	0 to 255
signed int	2 bytes	-32768 to +32767
unsigned int	2 bytes	0 to 65535
signed long	4 bytes	-2147483648 to +2147483647
unsigned long	4 bytes	0 to 4294967295
float	4 bytes	$\pm 1,176\text{E}-38$ to $\pm 3,40\text{E}+38$
double	8 bytes	$\pm 1,7\text{E}-308$ to $\pm 1,7\text{E}+308$
pointer	2 / 4 bytes	Address of objects
<b>Data Types for SFR access:</b>		
sbit	1 bit	0 or 1
sfr	2 bytes	0 to 65535

The **sbit** and **sfr** data types are included to allow access to the special function registers that are available on the 166. For example, the declaration: `sfr P2 = 0xFFC0;` declares the variable `P2` and assigns it the special function register address of `0xFFC0`. This is the address of PORT 2 on the C167.

## Memory Types

The C166 compiler explicitly supports the architecture of all 166/ST10 derivatives completely. It has full access to all hardware components of the 166 system. Each variable can be explicitly assigned to various memory types. Accessing the integrated RAM (on-chip RAM = **idata**) is considerably faster than accessing off-chip (**near**, **far**, **huge**, or **xhuge**) memory. Therefore it is useful to place often-used variables into on-chip memory, and to locate larger and less often accessed data elements into the **far** or **huge** memory.

3

Memory Type	166/ST10 Address Space
<b>near</b>	16 bit pointer; 16 bit address calculation allows access to: <ul style="list-style-type: none"> <li>• 16 KB for variables in NDATA group,</li> <li>• 16 KB for constants in NCONST group,</li> <li>• 16 KB for system area in SDATA group.</li> </ul> Together with the L166 directive DPPUSE you can have even up to 64KB NDATA/NCONST group. If <b>near</b> is applied to a function a CALLA or CALLR (16 bit call) is generated to this function.
<b>idata</b>	on-chip RAM of the 166; fastest access to variables.
<b>bdata</b>	bit-addressable on-chip RAM of the 166; supports mixed bit and byte accesses (limited to 256 bytes).
<b>sdata</b>	SYSTEM area (address space 0C000h-0FFFFh); this can be used for the definition of PEC addressable objects.
<b>far</b>	32 bit pointer; 16 bit address calculation allows full access to the whole address space. The size of a single object (array or structure) is limited to 16 KB. If <b>far</b> is applied to a function a CALLS (segmented call) instruction is generated to this function. <b>far</b> is the optimum memory type on the 80C166 CPU for accessing the 256KB address space of this CPU.
<b>huge</b>	32 bit pointer; 16 bit address calculation supports objects which size up to 64 KB. <b>huge</b> is the optimum memory type on the newer 166 derivatives like the C167 for accessing the 16MB address space
<b>xhuge</b>	32 bit pointer; 32 bit address calculation supports objects with unlimited size.

### Examples of Variable Declarations with Memory Type:

The following examples illustrate the use of the memory type within variable declarations:

```
char idata var1;
static unsigned long far array [100];
extern float near x, y, z;
extern unsigned int xhuge vector[50][100];
unsigned char sdata pec_buffer [100];

char bdata flags;
sbit flag0 = flags^0;
```

If the memory type is omitted in a variable declaration, the default or implicit memory type is selected. The default memory type depends on the memory model. For instance if a definition is made as **char var1[10]**, the default memory model **SMALL**, would define var1 to exist in near memory. If the **HLARGE** memory model is selected the **var1** would have been placed into **huge** memory.

## Memory Models

C166 supports seven memory models. With the exception of the **TINY** model, the 166/ST10 operates always in the **SEGMENTED** mode. The memory model determines the default memory type to be used for variable or function declarations without explicit memory type. With an explicit memory type the limits of the memory model in used can be by-passed. In the same way the access to variables can be speed-up, if for example the memory type **near** is used in the **LARGE** memory model.

Memory Model	Default Memory Type For...	
	Variables	Functions
<b>TINY</b>	<b>near</b>	<b>near</b> (up to 64KB code size)
<b>SMALL</b>	<b>near</b>	<b>near</b> (up to 64KB code size)
<b>COMPACT</b>	<b>far</b>	<b>near</b> (up to 64KB code size)
<b>HCOMPACT</b> (not for 8xC166 CPU)	<b>huge</b>	<b>near</b> (up to 64KB code size)
<b>MEDIUM</b>	<b>near</b>	<b>far</b> (unlimited code size)
<b>LARGE</b>	<b>far</b>	<b>far</b> (unlimited code size)
<b>HLARGE</b> (not for 8xC166 CPU)	<b>huge</b>	<b>far</b> (unlimited code size)

## Pointer

The memory type **near**, **far**, **huge** and **xhuge** can also be applied to pointers. A **near** pointer allows the accessing of all objects which are user stack based or defined in the **near**, **sdata**, **idata** or **bdata** area. A **far** pointer can access all objects in the 16MB address space, whereby the size of a single object is limited to 16KB. The memory type **huge** allows to access objects up to 64KB size. With **xhuge** very large objects with unlimited size can be accessed.

### *Note*

*Pointer arithmetic on huge pointers modifies only the 16-bit offset of the pointer. If you want to use a pointer to access the whole 16MB address space you must define a **xhuge** pointer.*

### Examples for using the memory type together with Pointers:

Variable Declaration	Pointer Size	Declaration of the Pointer
char c;	16/32 bit	char *ptr; (Pointer size depends from the memory model in use.)
int near nc;	16 bit	int near *np;
unsigned long far l;	32 bit	long far *lp;
char huge hc;	32 bit	char huge *hc_ptr;
float xhuge xf;	32 bit	char xhuge *xf_ptr;
void near func1 (void);	16 bit	void (near *fp1) (void);
int far func2 (void);	32 bit	int (far *fp2) (void);

## 3

### Registerbanks

C166 supports up to 128 logical register banks. Register banks can be used for example in connection with interrupt procedures. The code generated by C166 including all library functions is fully reentrant and independent from the register bank currently selected. This allows that the main program and one or more interrupt service routines can call simultaneously the same function.

### Interrupt Functions

The C166 compiler gives you complete control over all aspects of interrupt and register bank usage. Such support allows the system programmer to create efficient effective interrupt procedures. The user need only be concerned with the interrupt and necessary register bank switch over operation, in a general and high level manner. The C166 compiler generates only the code necessary to effect the most expedient handling. Refer to “Interrupt” on page 131 for example of an interrupt function.

### PEC Support

The **P**eripheral **E**vent **C**ontroller (PEC) can be directly programmed in the C166 source. Areas for PEC data can be define with the memory type **sdata** or must be explicitly located to SEGMENT 0 of the 166/ST10 memory. Refer to “Peripheral Event Controller” on page 134 for an example of using the PEC.

## Parameter Passing

Up to five parameters can be passed via CPU registers. This yields an efficient parameter passing which compares to assembler programming. If all five registers are used for parameters, the user stack is used for parameters. The user stack holds also automatic variables and is accessed via the R0 register (user stack pointer).

The return of function values takes place in fixed CPU registers, as listed in the table below. In this way the interface to assembler subroutines is very easy.

Return value	Register	Description
<b>bit</b>	R4.0	
<b>(unsigned) char</b>	RL4	
<b>(unsigned) int</b>	R4	
<b>(unsigned) long</b>	R4, R5	LSB in R4, MSB in R5
<b>float</b>	R4, R5	32 bit IEEE format, <b>exponent</b> and <b>sign</b> in R5
<b>double</b>	R4 to R7	64 bit IEEE format, <b>exponent</b> and <b>sign</b> in R7
<b>near *</b>	R4	
<b>far * or huge *</b>	R4, R5	Offset in R4, Selector in R5

## Code Optimizing

C166 optimizes the generated code with modern optimization techniques. The user has the choice of eight **OPTIMIZE** levels. In addition the type of code generation can be influenced with **OPTIMIZE (SIZE)** and **OPTIMIZE (SPEED)**. All optimizations executed by C166 are summarized below:

### General Optimizations

- Constant Folding
- Jump Optimizing
- Dead Code Elimination
- Register Variables
- Parameter Passing Via Registers
- Global And Local Common Sub-expression Elimination
- Strength Reduction
- Loop Rotation
- Dead Code Elimination
- Common Tail Merging

## 166/ST10 Specific Optimizations

- Peephole Optimization
- **NOP** and **DPP** Load Optimization
- Case/Switch Optimization

## Program Invocation

Typically, the C166 compiler will be called from the  $\mu$ Vision2 IDE when you build your project. However, you may invoke the compiler also within a DOS box by typing C166 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

```
>C166 MODULE.C COMPACT PRINT (E:M.LST) DEBUG SYMBOLS
C166 COMPILER V4.00

C166 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 C166 directives refer to “C166 Optimizing C Cross Compiler Directives” on page 166.

## Sample Program

The following example shows some functional capabilities of C166. The C166 compiler produces object files in OMF-166 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  $\mu$ Vision2 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 C166 compiler is shown on the next page.



C166 COMPILER V4.00, SAMPLE 12/01/99 10:31:08 PAGE 1

C166 COMPILER V4.00, COMPILATION OF MODULE SAMPLE  
 OBJECT MODULE PLACED IN SAMPLE.OBJ  
 COMPILER INVOKED BY: C:\KEIL\C166\BIN\C166.EXE SAMPLE.C CODE DEBUG

```

stmt level   source
1           #include <reg166.h>      /* register definitions for 80166 CPU */
2           #include <stdio.h>      /* standard i/o definitions      */
3
4           /* Convert to Upper Character */
5           unsigned char toupper (unsigned char c) {
6 1         if (c < 'a' || c > 'z') return 0;
7 1         else return (c & ~0x20);
8 1         }
9
10          sbit p310 = P3^10;      /* Port 3.10 output latch */
11          sbit dp310 = DP3^10;    /* Port 3.10 direction control register */
12          sbit dp311 = DP3^11;    /* Port 3.11 direction control register */
13
14          /* Initialize the Serial Interface 0 */
15          void init_serial (void) {
16 1         p310 = 1;              /* set Port 3.10 output latch (TxD) */
17 1         dp310 = 1;            /* set Port 3.10 direction control (TxD output) */
18 1         dp311 = 0;           /* reset Port 3.11 direction control (RxD input) */
19 1         S0TIC = 0x80;        /* set transmit interrupt flag */
20 1         S0RITC = 0x00;       /* delete receive interrupt flag */
21 1         S0BG = 0x40;         /* set baudrate to 9600 baud */
22 1         S0CON = 0x8011;      /* set serial mode */
23 1         }
24
25          /* Echo Upper Characters */
26          main () {
27 1         unsigned char c, buf[10];
28 1
29 1         init_serial ();
30 1
31 1         while (1) {
32 2         P0 = P2;              /* output hardware switch from Port2 */
33 2
34 2         gets (buf, sizeof (buf)); /* get input line */
35 2         for (c = 0; buf[c] != 0; c++) {
36 3             buf[c] = toupper (buf[c]); /* convert to capital */
37 3         }
38 2         printf ("%s\n", buf); /* echo input line */
39 2         P0 = 0;              /* clear Output Port to signal ready */
40 2     }
41 1     }

```

#### ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION toupper (BEGIN RMASK = @0x0030)
; SOURCE LINE # 5
;---- Variable 'c' assigned to Register 'R8' ----
; SOURCE LINE # 6

0000 F048      MOV     R4,R8
0002 C085      MOVBZ  R5,RL4
:
:

```

```

MODULE INFORMATION:  INITIALIZED  UNINITIALIZED
CODE SIZE           =           130  -----
NEAR-CONST SIZE     =             4  -----
FAR-CONST SIZE      =  -----
NEAR-DATA SIZE      =  -----
FAR-DATA SIZE       =  -----
IDATA-DATA SIZE     =  -----
SDATA-DATA SIZE     =  -----
BDATA-DATA SIZE     =  -----
BIT SIZE            =  -----
INIT'L SIZE         =  -----
END OF MODULE INFORMATION.

```

C166 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 C166 **CODE** directive. This gives you an assembly listing file with embedded source line numbers.

A memory overview provides information about the occupied 166/ST10 memory areas.

The number of errors and warnings in the

```
C166 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

*program are  
included at the end  
of the listing.*

## A166 Macro Assembler

A166 is a macro assembler for the 166/ST10 microcontroller family. A166 translates symbolic assembler language mnemonics into executable machine code. A166 allows you to define each instruction in a 166/ST10 program and is used where utmost speed, small code size and exact hardware control is essential. The A166 macro facility saves development and maintenance time, since common sequences need only be developed once.

### 3

## Source-Level Debugging

A166 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  $\mu$ Vision2 Debugger or emulators.

## Functional Overview

A166 translates an assembler source file into a relocatable object module. A166 generates a listing file, optionally with symbol table and cross reference. A166 contains two macro processors:

- **Standard Macros** are simple to use and enable you to define and to use macros in your 166/ST10 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 ASM86 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 A166 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.

```
A166 MACRO ASSEMBLER SAMPLE 24/01/99 15:44:45 PAGE 1
A166 MACRO ASSEMBLER V4.00
OBJECT MODULE PLACED IN SAMPLE.OBJ
INVOKED BY: C:\KEIL\C166\BIN\A166.EXE SAMPLE.A66 SET(SMALL) XREF DEBUG
```

```
LOC OBJ LINE SOURCE
1 $SEGMENTED
2
3 $IF MEDIUM OR LARGE
Model LIT 'FAR'
$ELSE
6 Model LIT 'NEAR'
7 $ENDIF
8
9 PUBLIC SERINIT, timerstop, timerstart
10 ASSUME DPP3:SYSTEM
11
12 ?PR?SERINIT section code
13
14 SERINIT proc NEAR
15
16 ;*****
17 ;** INIT SERIAL INTERFACE 0 **
18 ;*****
19
0000 AFE2 20 BSET P3.10 ; OUTPUT LATCH (TXD)
0002 AFE3 21 BSET DP3.10 ; DIR-CONTROL (TXD OUTPUT)
0004 BEE3 22 BCLR DP3.11 ; DIR-CONTROL (RXD INPUT)
0006 E7B68000 23 MOVB S0TIC,#080H ; TRANSMIT INTERRUPT FLAG
000A E7B70000 24 MOVB S0RIC,#000H ; RECEIVE INTERRUPT FLAG
000E E65A4000 25 MOV SOBG,#0040H ; 9600 BAUD
0012 E6D81180 26 MOV SOCON,#8011H ; SET SERIAL MODE
0016 CB00 27 RET
28 SERINIT endp
29
30
31 timerstart proc NEAR
0018 E6A00000 32 MOV T2CON,#0
001C E6A10000 33 MOV T3CON,#0
0020 E6200000 34 MOV T2,#0
0024 E6210000 35 MOV T3,#0
0028 E6A14000 36 MOV T3CON,#0040H
002C E6A04F00 37 MOV T2CON,#004FH
0030 CB00 38 RET
39 timerstart endp
40
41
42 timerstop proc NEAR
0032 E6A00000 43 MOV T2CON,#0
0036 E6A10000 44 MOV T3CON,#0
003A F2F442FE 45 MOV R4,T3
003E F2F540FE 46 MOV R5,T2
0042 CB00 47 RET
48 timerstop endp
49
50 ?PR?SERINIT ends
51
52 end
```

```
A166 MACRO ASSEMBLER SAMPLE 24/01/99 15:44:45 PAGE 2
```

#### XREF SYMBOL TABLE LISTING

```
-----
```

N A M E	TYPE	VALUE	I	ATTRIBUTES
?PR?SERINIT . . . . .	----	----		SECTION 12# 50
DP3 . . . . .	WORD	FFC6H	A	SFR 21 22
DPP3 . . . . .	WORD	FE06H	A	SFR 10
MODEL . . . . .	LIT	"NEAR"		6#
:	:	:	:	:
:	:	:	:	:

A166 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.

A166 is procedure oriented. All CPU instructions need to be placed with **PROC / ENDP** statements.

The listing contains a source line number and the object code generated by each source line.

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

Enable under **µVision2 Options for Target – Listing – Cross Reference** to get a detailed listing of all symbols used in the assembler source file.

```

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

```

*The number of errors and warnings in the program are included at the end of the listing..*

## L166 Linker/Locator

The L166 linker/locator combines several program modules into one executable 166/167 program. In doing so the external and public references are resolved and the relocatable program parts are assigned to absolute addresses. The modules to be combined may have been written in C or assembler. L166 automatically selects the appropriate run-time libraries and links only the library modules that are required.

3

### Address Management

The C166 compiler assigns each code and data section to a specific class name. The class name refers to the different memory areas; for example the class **NCODE** contains code which must be directed to ROM space; the class **NDATA** contains variable sections which must be directed to RAM space.

During the link/locate process all sections with the same class name are located to a specific memory area. The  $\mu$ Vision2 IDE delivers the correct settings for L166 **CLASSES** directive from the selected CPU and specifications under **Options – Target** for *external memory* and *on-chip* memory components.

```
CLASSES (FCODE (0x10000 - 0x1FFFF, 0x40000 - 0x5FFFF))
```

Above is an example for an L166 **CLASSES** directive. This instructs L166 to use the address spaces 0x10000 - 0x1FFFF and 0x40000 – 0x5FFFF for the **FCODE** class (= far code). L166 locates all sections with the class name **FCODE** to this memory region, which is usually ROM space. You can enter address ranges for user defined memory classes under **Options – L166 Locate – Users Classes**.

Exact placing of a section is also possible; enter in  $\mu$ Vision2 dialog **Options – L166 Locate – User Sections** the address specification for individual sections. For example, **?XD?MYPROG%XDATA (0x20000)** in this dialog field will insert the L166 **SECTIONS** directive which locates the section with the name **?XD?MYPROG** and the memory class **XDATA** to address 0x20000:

```
SECTIONS (?XD?MYPROG%XDATA (0x20000))
```

## Map File

On the following page is an example listing file generated by L166.

L166 LINKER/LOCATER V4.00 22/01/99 10:32:02 PAGE 1

L166 LINKER/LOCATER V4.00, INVOKED BY:  
C:\KEIL\C166\BIN\L166.EXE SAMPLE.OBJ

CPU TYPE: C166  
CPU MODE: SEGMENTED  
MEMORY MODEL: SMALL

INPUT MODULES INCLUDED:  
SAMPLE.OBJ (SAMPLE)  
COMMENT TYPE 128: C166 V4.00  
C:\KEIL\C166\LIB\C166S.LIB (?C\_STARTUP)  
COMMENT TYPE 128: A166 V4.00  
C:\KEIL\C166\LIB\C166S.LIB (PRINTF)  
COMMENT TYPE 128: A166 V4.00  
:  
:

INTERRUPT PROCEDURES OF MODULE: SAMPLE (SAMPLE)

INTERRUPT PROCEDURE	INT	INTERRUPT NAME
C_STARTUP	0	RESET

MEMORY MAP OF MODULE: SAMPLE (SAMPLE)

START	STOP	LENGHT	TYPE	ALIGN	TGR	GRP	COMB	CLASS	SECTION NAME
00000H	00003H	00004H	---	---	---	---	---	* INTVECTOR TABLE *	
00004H	00085H	00082H	CODE	WORD	---	1	PUBL	NCODE	?PR?SAMPLE
00086H	0044FH	003CAH	CODE	WORD	---	1	PRIV	NCODE	?PR?printf
00450H	004BDH	0006EH	CODE	WORD	---	1	PUBL	NCODE	?PR?GETS
:	:	:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:	:	:

GROUP LIST OF MODULE: SAMPLE (SAMPLE)

GROUP NAME	TYPE	TGR	GRP	CLASS	SECTION NAME
NCODE	CODE	---	1	NCODE	?PR?SAMPLE
				NCODE	?PR?printf
				NCODE	?PR?GETS
:	:	:	:	:	:
:	:	:	:	:	:

PUBLIC SYMBOLS OF MODULE: SAMPLE (SAMPLE)

VALUE	PUBLIC SYMBOL NAME	REP	TGR	CLASS	SECTION NAME
00608H	?C_CLRMEMSECSTART		VAR	---	--
00606H	?C_INITSECSTART		VAR	---	--
00000H	?C_PAGEDPP1		CONST	---	--
00001H	?C_PAGEDPP2		CONST	---	--
:	:	:	:	:	:
:	:	:	:	:	:

SYMBOL TABLE OF MODULE: SAMPLE (SAMPLE)

VALUE	TYPE	REP	LENGTH	TGR	SYMBOL NAME
0003AH	GLOBAL	LABEL	---	---	main
00022H	PUBLIC	LABEL	---	---	init_serial
00004H	PUBLIC	LABEL	---	---	toupper
00004H	BLOCK	LVL=0	001EH	---	toupper
00008H	SYMBOL	REG	---	---	c
00004H	LINE	---	---	---	#5
00004H	LINE	---	---	---	#6
00018H	LINE	---	---	---	#7
00020H	LINE	---	---	---	#8
---	BLOCKEND	LVL=0	---	---	
:	:	:	:	:	:
:	:	:	:	:	:

L166 produces a MAP file (extension .M66) with date and time of the link/locate run.

L166 displays the invocation line, memory model, CPU type, and CPU mode.

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

All interrupt procedures with assigned trap numbers are listed.

The memory map contains the usage of the physical 80C166 memory.

The Group list shows the group, class and section names used in the application program.

All public symbols together with their values are listed.

A complete list of all debug symbols is printed.

Warning messages and error messages are listed at the end of the MAP file. These may point to

```
L166 RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

*possible problems  
encountered during  
the link/locate run.*

## LIB166 Library Manager

The LIB166 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 which may be accessed by the L166 linker/locator.

To build a library with the  $\mu$ Vision2 project manager enable **Options for Target – Output – Create Library**. You may also call **LIB166** from a DOS box. Refer to “LIB166 Library Manager Commands” on page 170 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.

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 L166 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.

## OH166 Object-HEX Converter

The OH166 object-HEX converter creates Intel HEX files from absolute object modules which are created by the L166 linker/locator. Intel HEX files are ASCII files that contain a hexadecimal representation of your application program. They can be easily loaded into a device programmer for programming EPROMS. Both HEX-86 (1 MB address range) and HEX-386 (16 MB address range) file formats are supported. You may also create HEX files to program FLASH memory devices. The data records in these files are sorted in ascending order. Unused bytes are filled with a specified byte value.



## 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 C166 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  $\mu$ Vision2 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 C166 compiler for optimum code quality, and the features of the  $\mu$ Vision2 project manager.

**4**

### Create a Project

$\mu$ Vision2 includes a project manager which makes it easy to design applications for the 166 family. You need to perform the following steps to create a new project:

- Start  $\mu$ Vision2, 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 166/ST10 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  $\mu$ Vision2 project.

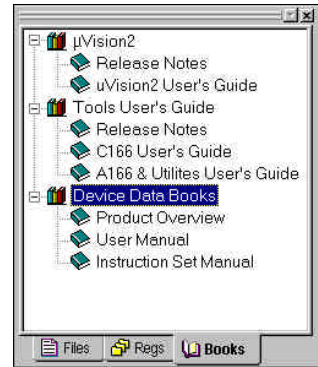
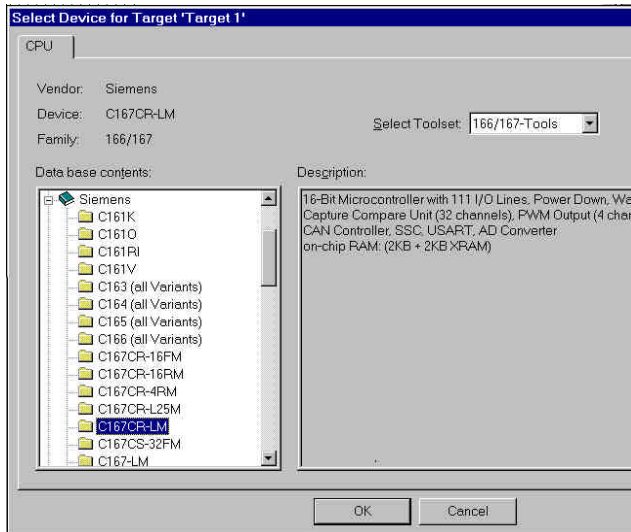


## Start $\mu$ Vision2 and Create a Project File

$\mu$ Vision2 is a standard Windows application and started by clicking on the program icon. To create a new project file select from the  $\mu$ Vision2 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**.  $\mu$ Vision2 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  $\mu$ Vision2 device database. Just select the microcontroller you use. We are using for our examples the Siemens C167CR-LM CPU. This selection sets necessary tool options for the C167CR-LM device and simplifies in this way the tool configuration.



### NOTE

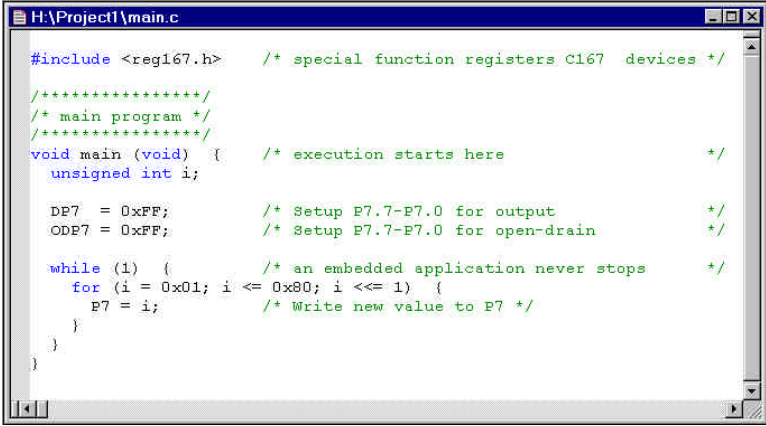
*On some devices, the  $\mu$ Vision2 environment needs additional parameters that you have to enter manually. Please carefully read the information provided under **Description** in this dialog, since it might have additional instructions for the device configuration.*

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.  $\mu$ Vision2 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 <reg167.h> /* special function registers C167 devices */

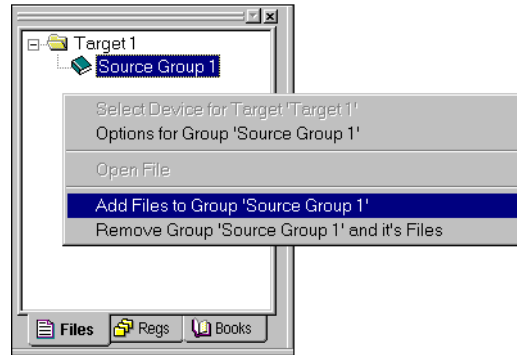
/*****
/* main program */
*****/

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

    DP7 = 0xFF; /* Setup P7.7-P7.0 for output */
    ODP7 = 0xFF; /* Setup P7.7-P7.0 for open-drain */

    while (1) { /* an embedded application never stops */
        for (i = 0x01; i <= 0x80; i <<= 1) {
            P7 = i; /* Write new value to P7 */
        }
    }
}
```

Once you have created your source file you can add this file to your project.  $\mu$ Vision2 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.



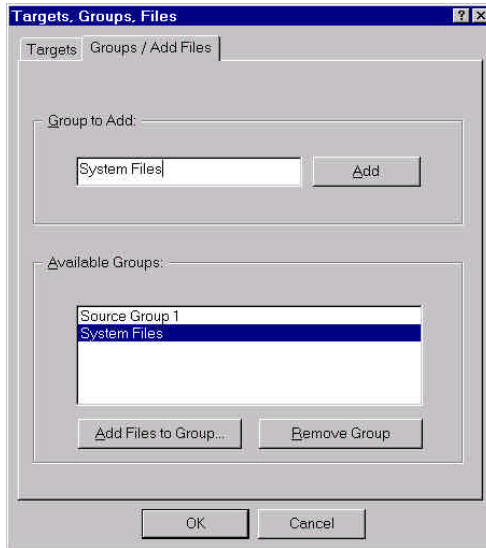
## Add and Configure the Startup Code

### 4

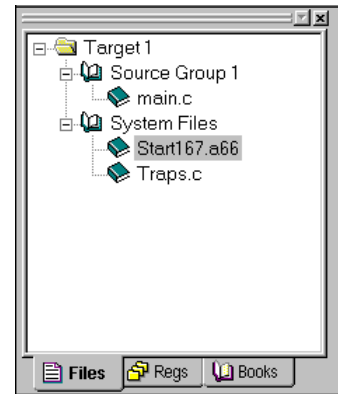
Typically, a 166/ST10 program requires a CPU initialization code that needs to match the configuration of your hardware design. For most 166 / ST10 derivatives, you should use the **START167.A66** file as startup code. Only the 8xC166 CPU variants are initialized with the **STARTUP.A66** file. Since you need to modify that file to match your target hardware, you should copy the **START167.A66** file from the folder **C:\KEIL\C166\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 **START167.A66** file to your project.

Another file that aids you in program debugging is **TRAPS.C**. This file contains service routines for the various CPU hardware traps (interrupts) which are called on hardware or software failures. Also **TRAPS.C** can be copied from the folder **C:\KEIL\C166\LIB** to your project folder and added in the same way.



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

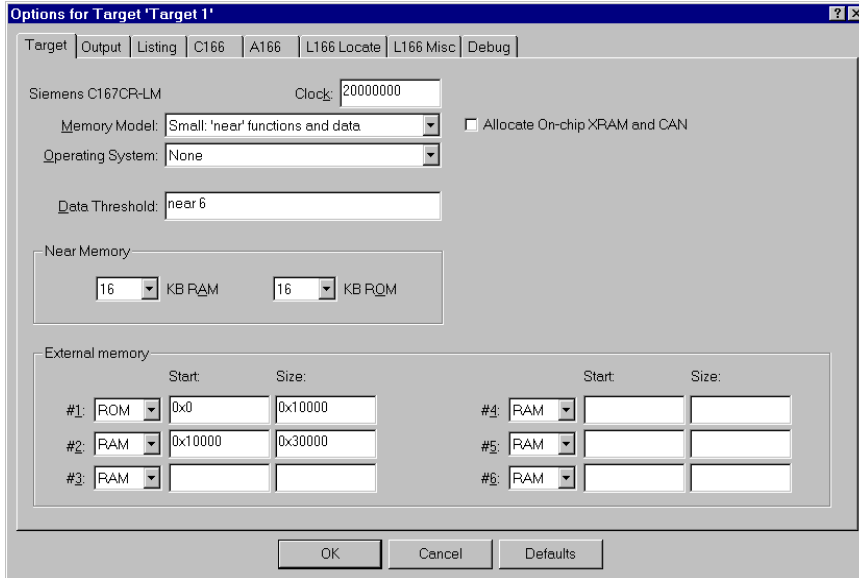


The  $\mu$  Vision2 **Project Window – Files** should now show the above file structure. Open **START167.A66** in the editor with a double click on the file name in the project window. Then you configure the startup code as described in “Chapter 10. CPU and C Startup Code” on page 153. It is very important that the settings in the startup code 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.



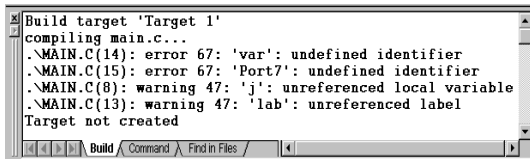
The following table describes the options of the **Target** dialog:

Dialog Item	Description
Clock	specifies the internal CPU clock of your device. Most 166 designs are using the on-chip PLL to generate the CPU clock, in most cases this value is not identical with the XTAL frequency. Check your hardware design carefully to determine the correct value.
Allocate On-chip ...	specifies the usage of the on-chip components which are typically enabled in the CPU startup code. Make sure that the dialog settings are identical with the START167.A66 settings.
Memory Model	specifies the C166 compiler memory model. For starting new applications the default <b>SMALL</b> is a good choice. Refer to "Memory Models and Memory Types" on page 48 for a discussion of the various memory models.
Data Threshold Near Memory	allows you to optimize the memory model settings. Refer to "Data Threshold" on page 50 for more information.

External Memory	here you specify all external memory areas of the target hardware. RAM denotes the memory areas where variables are stored. ROM refers to areas that store constants and program code (typical EPROM or Flash memory). When using the Monitor-166, the program will run in RAM space. However you have to specify ROM in this dialog, otherwise your application has no memory for constants and program code. For more information, refer to “Target Options when Using Monitor-166” on page 159.
-----------------	--

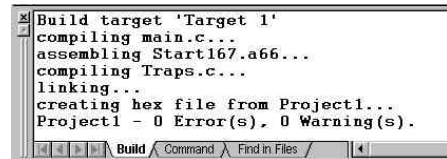
## Build Project and Create a HEX File

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,  $\mu$ Vision2 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  $\mu$ Vision2 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
  
```

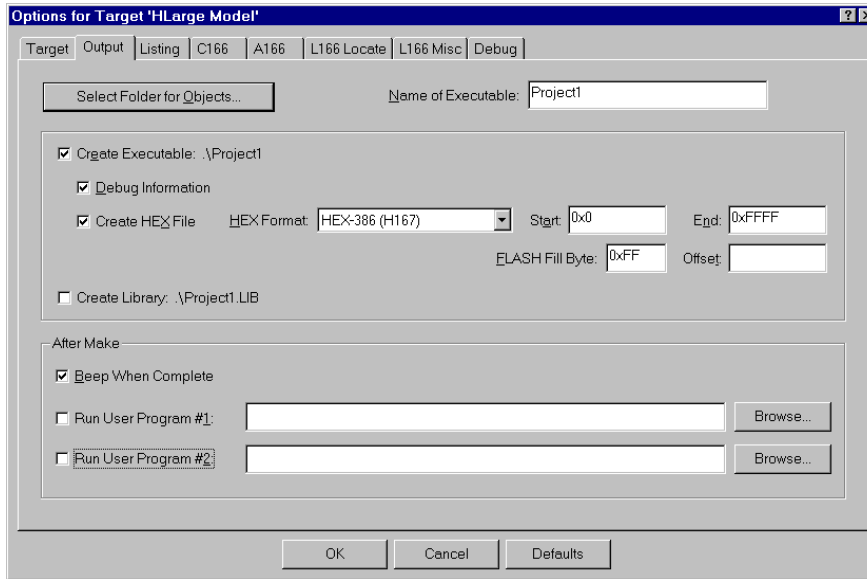


```

Build target 'Target 1'
compiling main.c...
assembling Start167.a66...
compiling Traps.c...
linking...
creating hex file from Project1...
Project1 - 0 Error(s), 0 Warning(s).
  
```

4

Once you have successfully generated your application you can start debugging. Refer to “Chapter 5. Testing Programs” on page 67 for a discussion of the  $\mu$ Vision2 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.  $\mu$ Vision2 creates HEX files with each build process when **Create HEX file** under **Options for Target – Output** is enabled. The **FLASH Fill Byte**, **Start** and **End** values direct the OH166 utility to generate a sorted HEX files; sorted files are required for some Flash programming utilities. You may start your PROM programming utility after the make process when you specify the program under the option **Run User Program #1**.



4

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.  $\mu$  Vision2 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  $\mu$  Vision2 rebuilds files only when needed. With the **Rebuild Target** command, all source files are translated, regardless of modifications.

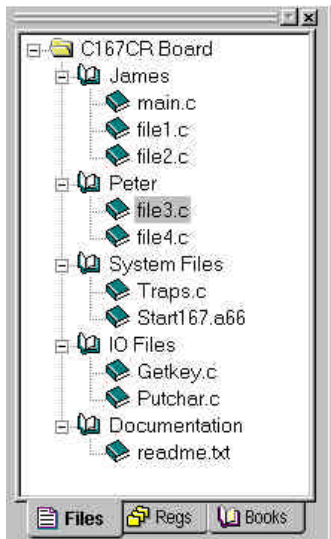
## Project Targets and File Groups

By using different **Project Targets**  $\mu$  Vision2 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  $\mu$  Vision2.

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.

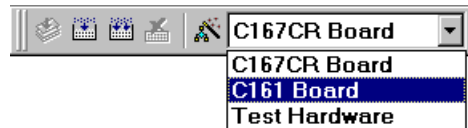




The **Project Windows** shows all groups and the related files. Files are built and linked in the same order as shown in this window. You can move file positions with **Drag & Drop**. You may select a target or group name and **Click** to rename it. The local menu opens with a right mouse **Click** and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

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

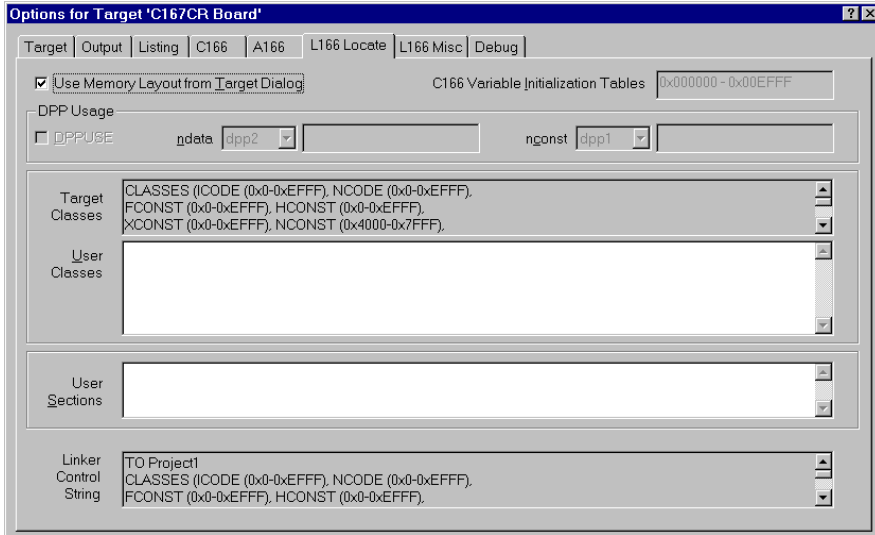
In the build toolbar you can quickly change the current target to build.



## 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 40 for details.
Output	Define the output files of the tool chain and allows you to start user programs after the build process. See page 56 for more information.
Listing	Specify all listing files generated by the tool chain.
C166	Set C166 compiler specific tool options like code optimization or variable allocation. Refer to “Other C166 Compiler Directives” on page 53 for information.
A166	Set assembler specific tool options like macro processing.
L166 Locate	Define the location of memory classes and sections. Typical you will enable <b>Use Memory Layout from Target Dialog</b> as show below to get automatic settings. Refer to “Locate Sections to Absolute Memory Locations” on page 60 and “User Classes” on page 61 for more information on this dialog.
L166 Misc	Other linker related settings like <b>Warning</b> or memory <b>Reserve</b> directive. You need to reserve some memory locations when you are using Monitor-166 for debugging. For more informatino refer to “Target Options when Using Monitor-166 “ on page 159 for more information.
Debug	Settings for the $\mu$ Vision2 Debugger. Refer to page 74 for more information.
Properties	File information and special options for files and groups refer to “File and Group Specific Options – Properties Dialog” on page 62.



## 4

## µVision2 Utilities

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



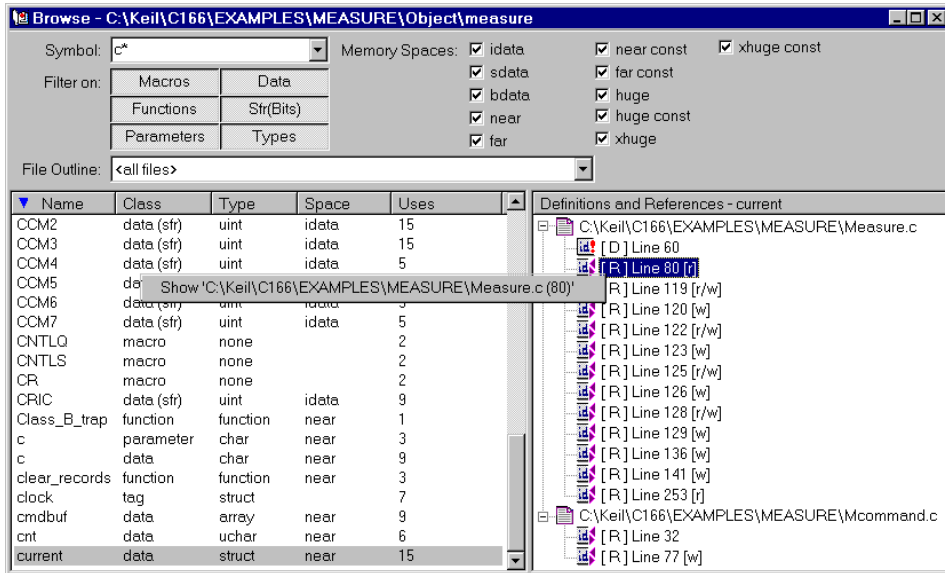
### Find in Files

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.



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:

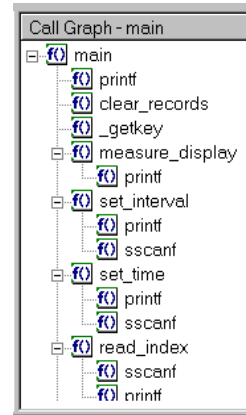
Browse Options	Description
Symbol	specify a mask that is used to match symbol names. The mask may consist of alphanumeric characters plus mask characters: # matches a digit (0 – 9) \$ matches any character * matches zero or more characters.
Filter on	select the definition type of the symbol
File Outline	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.
<u>a</u> \$#*	... with an underline, followed by the letter <b>a</b> , followed by any character, followed by a digit, ending with zero or more characters. For example, <b><u>ab1</u></b> or <b><u>a10value</u></b> .
<u>*ABC</u>	... with an underline, followed by zero or more characters, followed by <b>ABC</b> .

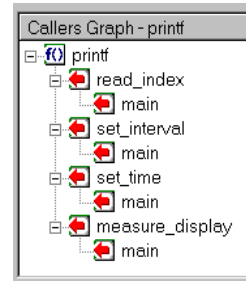
The local menu in the Browse window opens with a right mouse Click and allows you to open the editor on the selected reference. For functions you can also view the Call and Callers graph. The Definitions and References view gives you additional information with the following symbols:

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



## 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...**. With this dialog you configure the parameters of the user applications.

Using a key sequence in the Parameters Line you may pass arguments from the  $\mu$ Vision2 project manager to these user programs. 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
\$	folder name of the file specified in the file code (C:\MYPROJECT)
#	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)

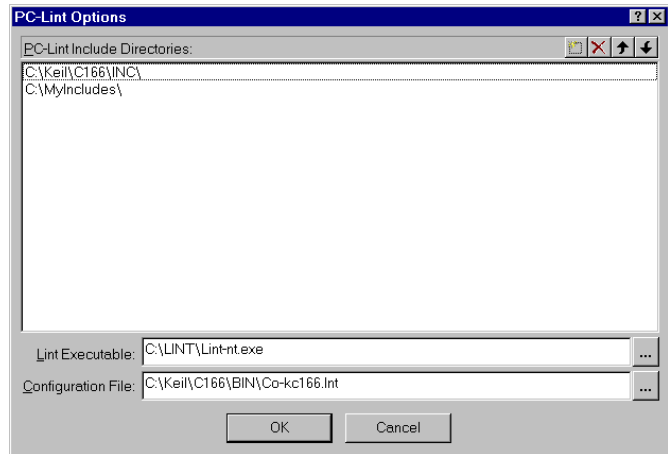
To use \$, #, %, or @ in the user program command line, use \$\$, ##, %%, or @@  
 For example @@ gives a single @ in the user program command line.

File Code	Specifies the file name inserted in the user program line
F	current in focus editor file. (MEASURE.C)
P	name of the current project file (PROJECT1.UV2)
X	µVision2 executable program file (C:\KEIL\UV2\UV2.EXE)
H	application HEX file (PROJECT1.H86)
L	linker output file, typical the executeable file for debugging (PROJECT1)

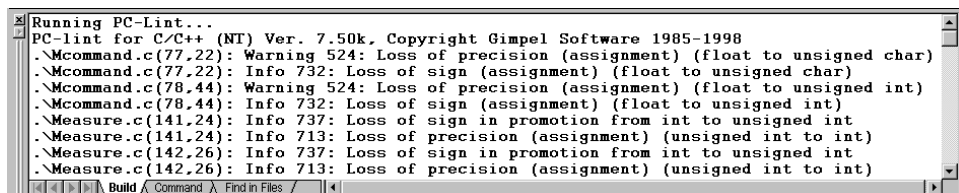
## Running PC-Lint

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.

You need to install **PC-Lint** on your PC and specify various parameters with the dialog **Tools – PC Lint Options**. The example shows a typical PC-Lint configuration. To get correct output in the **Build** page, you need to use the configuration file that is located in the folder **KEIL\C166\BIN**.



After the setting the PC-Lint options, 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.



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

The configuration file `C:\KEIL\C166\BIN\CO-KC166.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 C166 compiler.

## Writing Optimum Code

Many configuration parameters have influence on the code quality of your 166 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.

# 4

### Memory Models and Memory Types

The most significant impact on code size and execution speed has the memory model. The memory model influences variable access and CALL/RET instructions. Refer to “Memory Models” on page 23 for detailed information. The following table allows you to determine the memory model that best fits your application. The memory model is selected in the **Options for Target – Target** dialog page.

Total Code Size of the Application	Data and Constant Size of the Application	
	less than 64KB	more than 64KB
less than 64 KB	<b>SMALL</b>	<b>HCOMPACT</b>
more than 64KB	<b>MEDIUM</b>	<b>HLARGE</b>

This table is valid for devices with extended instruction set (i.e. C161, C163, C164, C165, C167, ST10-272, ST10-262) and SEGMENTED CPU mode. This is typical for all new applications.

### Tips for SMALL and MEDIUM Memory Model

Even if your application exceeds the 64KB data/constant limit you may still use the **SMALL** or **MEDIUM** memory model. To bypass the 64KB variable limitation, you may direct some variables to **huge** or **xhuge** memory. However, the address of such variables cannot be passed to C run-time library functions, like **printf** or **scanf**. The program will not work since C run-time library functions cannot access **huge** or **xhuge** variables in this memory models. The

**HCOMPACT** and **HLARGE** memory model does not have this limitation and should be used instead. Another alternative is, that you copy variables to near memory and pass the address of the near variable to the C library function.

## Tips for HCOMPACT and HLARGE Memory Model

You may apply **near** memory type to optimize variable accesses in this memory model. This generates 16-bit addressing instead of 32-bit addressing. Frequently used variables should be located in the 166 on-chip memory with the **idata** memory type. The C166 compiler **HOLD** directive may direct variables below a specified size automatically to optional memory areas. For more information refer to “Data Threshold” on page 50.

---

### NOTE

*The **far** memory type is provided for compatibility to the 8xC166 CPU and support of existing C166 code. For new designs you should use **huge** instead of **far**, since this memory type has an object size limit of 64KB and generates better code for near pointer cast operations. This is why you should use **HCOMPACT** instead of **COMPACT** or **HLARGE** instead of **LARGE** when you select a memory model.*

---

## Bit-field Structures

C166 accesses bit-field struct members with size 1 bit directly with CPU BIT instructions if the struct is located in the **bdata** space. You can also enter a **Data Threshold** in the **Options for Target – Target** dialog to locate such bit-field structures automatically to the **bdata** space. Refer also to “Data Threshold” later in this section. The following shows an example:

```

struct test {
    int    bit0: 1;           // this structure contains some
    int    bit1: 1;           // fields with size 1 bit
    int    bit2: 1;
    int    value1: 4;
    int    value2: 9;
};

struct test bdata t;        // locate to bit-addressable space

void main (void) {
    t.bit0 = 1;              // generates BSET instruction
    if (t.bit1) t.bit2 = t.bit0; // uses JB and BMOV instructions
}

```

## Data Threshold

In the **Options for Target – Target** dialog you may enter a **Data Threshold** to specify the default memory type for small objects. This entry generates directly the C166 HOLD directive. If the **LARGE, HLARGE** or **COMPACT, HCOMPACT** memory model is used together with the default **near, 6** all variable definitions without explicit memory type which occupied not more then 6 bytes are place in the **near** area. Objects that require more then 6 Bytes are located in the **far** or **huge** area.

### Examples:

Memory Model: HCompact: 'huge' data, 'near' funcs  
 Operating System: None  
 Data Threshold: sdata 10

Locates variables with size < 10 Bytes to **sdata** space. This is typically the XRAM space in a 166 device. Other variables without explicit memory space are in the **huge** space.

Memory Model: HLarge: 'huge' data, 'far' funcs  
 Operating System: None  
 Data Threshold: near 6, idata 2, bdata 2

Locates variables with size < 2 Bytes to **idata**, bit-field structs with single bit members to **bdata** and variables with size < 6 Bytes to **near**. Other variables without memory type are in **huge**.

### NOTE

*The C166 HOLD directive should be identical for all modules in an application. A problem may arise when a module with an extern variable definition is translated with a different data threshold setting than the module defining this variable. In this case, the C166 compiler might have different memory type settings for this variable. When you create libraries with global variables or accesses to extern application variables, you should use the same data threshold setting for the application using this library. As an alternative, you may specify explicit memory types.*

## Global Register Optimization

The Keil 166 tools provide support for application wide register optimization which is enabled in the Options for Target – C166 dialog with **Global Register Coloring**. With the application wide register optimization, the C166 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  $\mu$ Vision2 build process makes automatically iterative re-translations of C source files.

In the following example the function *func1* calls the external function *func2*. *func2* is using not all the CPU registers. Therefore *func1* can use some CPU register without saving it. Due to the global register optimization, the C166 compiler is aware of this situation and can take advantage of it.

## With Global Register Optimization

```

char *func1 (void) {
    int a,b, c;
    char *s;

    b = c = 0;
    MOV     R1,#0
;---- Variable 'c' assigned to 'R1'
    MOV     R2,#0
;---- Variable 'b' assigned to 'R2'
    for (a = 0; a < 100; a++) {
        MOV     R12,#0
;---- Variable 'a' assigned to 'R12'
?C0004:
        c += (b + 1);
        MOV     R4,R2
        ADD     R4,#1
        ADD     R1,R4
        s = func2 ();
        CALL    func2
        MOV     R6,R4
        MOV     R7,R5
;---- Variable 's' assigned to 'R6/R7'
        b = strlen (s);
        MOV     R9,R5
        MOV     R8,R4
        CALL    strlen
        MOV     R2,R4
    }
    CMPIL   R12,#063H
    JMPR    cc_SLT,?C0004
    val = c;
    MOV     val,R1
    return (s);
    MOV     R4,R6
    MOV     R5,R7
    RET
}

void func0 (void) {
    int j;

    for (j = 0; j < 100; j++) {
        MOV     R3,#0
;---- Variable 'j' assigned to 'R3'
?C0010:
        func1 ();
        CALL    func1
    }
    CMPIL   R3,#099
    JMPR    cc_SLT,?C0010
}

    RET

```

Code Size: 60 Bytes

## Without Global Register Optimization

```

char *func1 (void) {
    int a,b, c;
    char *s;

    PUSH    R13
    PUSH    R14
    PUSH    R15
    SUB     R0,#4

    b = c = 0;
    MOV     R14,#0
;---- Variable 'c' assigned to 'R14'
    MOV     R15,#0
;---- Variable 'b' assigned to 'R15'
    for (a = 0; a < 100; a++) {
        MOV     R13,#0
;---- Variable 'a' assigned to 'R13'
?C0004:
        c += (b + 1);
        MOV     R4,R15
        ADD     R4,#1
        ADD     R14,R4
        s = func2 ();
        CALL    func2
        MOV     [R0],R4           ; s
        MOV     [R0+#02H],R5     ; s+2
    }
    b = strlen (s);
    MOV     R9,[R0+#02H]       ; s+2
    MOV     R8,[R0]           ; s
    CALL    strlen
    MOV     R15,R4
    }
    CMPIL   R13,#063H
    JMPR    cc_SLT,?C0004
    val = c;
    MOV     val,R14
    return (s);
    MOV     R5,[R0+#02H]       ; s+2
    MOV     R4,[R0]           ; s
    RET
}

void func0 (void) {
    int j;
    PUSH    R13
    for (j = 0; j < 100; j++) {
        MOV     R13,#0
;---- Variable 'j' assigned to 'R3'
?C0010:
        func1 ();
        CALL    func1
    }
    CMPIL   R13,#099
    JMPR    cc_SLT,?C0010
}

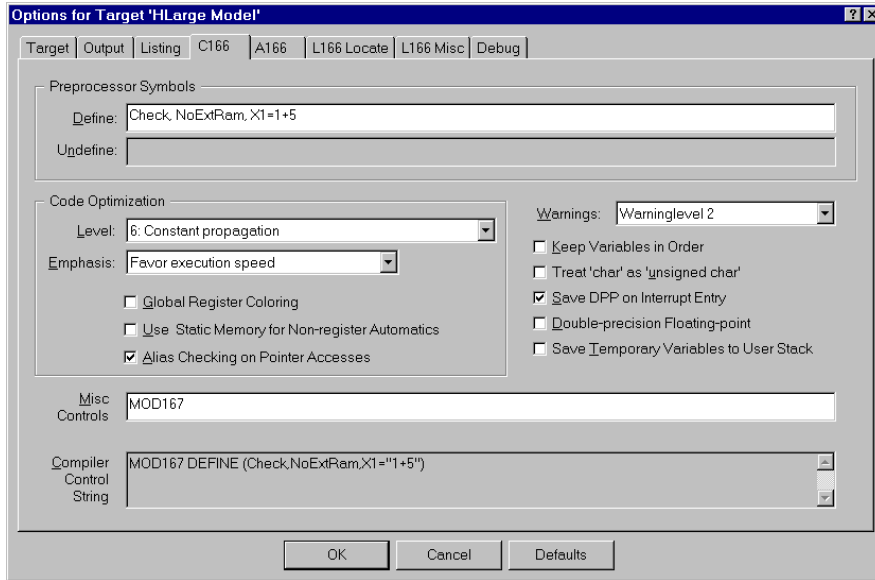
    POP     R13
    RET

```

Code Size: 86 Bytes

## Other C166 Compiler Directives

There are several other C166 directives that improve the code quality. These directives are enabled in the Options – C166 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.



The following table describes the options of the **C166** dialog page:

Dialog Item	Description
Define	outputs the C166 <b>DEFINE</b> directive to enter preprocessor symbols.
Undefine	is only available in the <b>Group</b> and <b>File Options</b> dialog and allows you to remove DEFINE symbols that are specified at the higher target or group level.
Code Optimization Level	specifies C166 OPTIMIZE level. Typical you will not alter the default. With the highest level "7: Common Tail Merging" the compiler analyzes the code and tries to locate common heads and tails. If the compiler detects common code sequences, it will replace one code sequence by a jump instruction to the other equivalent code sequence. While analyzing the code, the compiler also tries to replace sequences with cheaper instructions. Since the compiler inserts JMP 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 C166 compiler inserts for some C operations (like long shift) loops instead of a faster replacement code. Also some intrinsic sequences (like struct copy) are replace by a library call.
Global Register Coloring	enables the "Global Register Optimization". Refer to page 50 for details.

Dialog Item	Description
Use Static Memory for Non-register Automatics	instructs C166 to use static memory locations for function automatics (not parameters) which cannot be allocated in CPU registers. The code generated is not longer reentrant, but code size and execution speed can be improved, since the CPU can access static variables faster.
Alias Checking on Pointer Accesses	when this option is disabled, C166 ignores pointer write operations during the optimization phase. If a CPU register holds a variable it gets reused, even when a pointer write could modify that variable.
Keep Variables in Order	tells the C166 compiler to order the variables in memory according their definition in the C source file. This option does not influence code quality.
Treat 'char' as 'unsigned char'	instructs C166 to treat all variables declared with plain <b>char</b> as <b>unsigned char</b> variables. This option does not influence code quality.
Save DPP on Interrupt Entry	instructs C166 not to save DPP registers on interrupt entries. This directive should not be used for the 8xC166 CPU and when you are sure that your assembler code does not alter the DPP registers in any way. The C run-time libraries and the RTX166 operating system never modifies these registers when you are generating code with extended instruction set (MOD167 directive used), this is typical for all new applications.
Double-Precision Floating-Point	When this option is disabled, C166 uses single precision floating point arithmetic even if the <b>double</b> keyword is used the C source file.
Save Temporary Variables to User Stack	directs the C compiler to save temporary results and saved-by-callee variables to the User Stack. The default setting saves temporary results to the System Stack; this is always fast on-chip RAM it is faster, but the size is limited.
Misc Controls	allows you to enter special C166 directives. You may need such options when you are using very new 166/ST10 devices with chip bugs, to enter compiler FIXxxx directives.
Compiler Control String	displays the C166 compiler invocation string. Allows you can verify the compiler options currently for your source files.

## Data Types

The 166/ST10 CPU is a 16-bit microcontroller. Operations that use 16-bit types (like **int** and **unsigned int**) are more efficient than operations that use char or long types. For this reason, it is typically better to use the **int** or **unsigned int** data type for all automatic and parameter variables in your application instead of **char** or **unsigned char**. This is exemplified in following code example:

Program with 'int' data type	Program with 'char' data type
<pre> char array[100];  char test (int i) {     int k;      for (k = 0; k &lt; 10; k++) {         array[k] = 0;     }     return (array[i]); } </pre>	<pre> char array[100];  char test (char i) {     char k;      for (k = 0; k &lt; 10; k++) {         array[k] = 0;     }     return (array[i]); } </pre>

Generated Code: 18 Bytes	Generated Code: 28 Bytes
<pre> ;---- Variable 'i' assigned to 'R8' MOV     R6,#00H ;---- Variable 'k' assigned to 'R6' MOV     RL5,#00H ?C0004: MOV     [R6+#array],RL5 CPI1   R6,#09H JMPR   cc_SLT,?C0004  MOV     RL4,[R8+#array] RET </pre>	<pre> ;---- Variable 'i' assigned to 'R8' MOVB   RL6,#00H ;---- Variable 'k' assigned to 'RL6' MOVB   RL5,#00H ?C0004: MOVBS  R4,RL6 MOVB   [R4+#array],RL5 ADDB  RL6,#01H CMPB  RL6,#0AH JMPR  cc_SLT,?C0004 MOV   R4,R8 MOVBS R4,RL4 RET </pre>

## Applications without external RAM Devices

For single-chip applications or hardware with no external RAM devices the User Stack and the System Stack size needs to be reduced in the CPU startup file. The following values in the startup files are good choices when only on-chip RAM is available in a system. Refer to “Configuring the Startup Code” on page 153 for more information.

```

$SET (STK_SIZE = 2)           ; for 64 words system stack size
USTSZ EQU 80H                ; for 128 bytes user stack size.

```

The C166 directive **USERSTACKDPP3** changes the assumption made by the C166 compiler regarding the access of the User Stack area. The default User Stack is allocated in the **NDATA** memory class and accessed with DPP2. The **USERSTACKDPP3** should be applied when DPP0, DPP1 and DPP2 are used to access the **NCONST** class. You enter this directive under **Misc Controls** in the **Options for Target – C166** dialog. If **USERSTACKDPP3** is used, the **?C\_USERSTACK** section definition must be modified in the **STARTUP.A66** or **START167.A66** file as shown below:

```

?C_USERSTACK SECTION DATA PUBLIC 'IDATA'

```

or

```

?C_USERSTACK SECTION DATA PUBLIC 'SDATA'

```

Also you need to change the line:

```

MOV     R0,#DPP2:?C_USERSTKTOP

```

to

```

MOV     R0,#DPP2:?C_USERSTKTOP

```

## Tips and Tricks

The following section discusses advanced techniques you may use with the  $\mu$ Vision2 project manager. You will not need the following features very often, but readers of this section get a better feeling for the  $\mu$ Vision2 capabilities.

### Import Project Files from $\mu$ Vision Version 1

You can import project files from  $\mu$ Vision1 with the following procedure:

1. Create a new  $\mu$ Vision2 project file and select a CPU from the device database as described on page 36. It is important to create the new  $\mu$ Vision2 project file in the existing  $\mu$ Vision1 project folder.
2. Select the old  $\mu$ Vision1 project file that exists in the project folder in the dialog **Project – Import  $\mu$ Vision1 Project**. This menu option is only available, if the file list of the new  **$\mu$ Vision2** project file is empty.
3. This command imports the old  $\mu$ Vision1 linker settings into the L166. But, we recommend that you are using the  $\mu$ Vision2 **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 – L166 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  $\mu$ Vision2 project file.
5. You may now create file groups in the new  $\mu$ Vision2 project as described under “Project Targets and File Groups” on page 42. Then you can **Drag & Drop** files into the new file groups.

---

#### NOTE

*It is not possible to make a 100% conversion from  $\mu$ Vision1 project files since  $\mu$ Vision2 differs in many aspects from the previous version. After you have imported your  $\mu$ Vision1 check carefully if the tool settings are converted correct. Some  $\mu$ Vision1 project settings, for example user translator, library module lists and special compiler and linker controls like the C166 PECDEF and L166 OBJECTCONTROLS directive are not converted to the  $\mu$ Vision2 project. Also the dScope Debugger settings cannot be copied to the  $\mu$ Vision2 project file.*

---

## Start External Tools after the Build Process

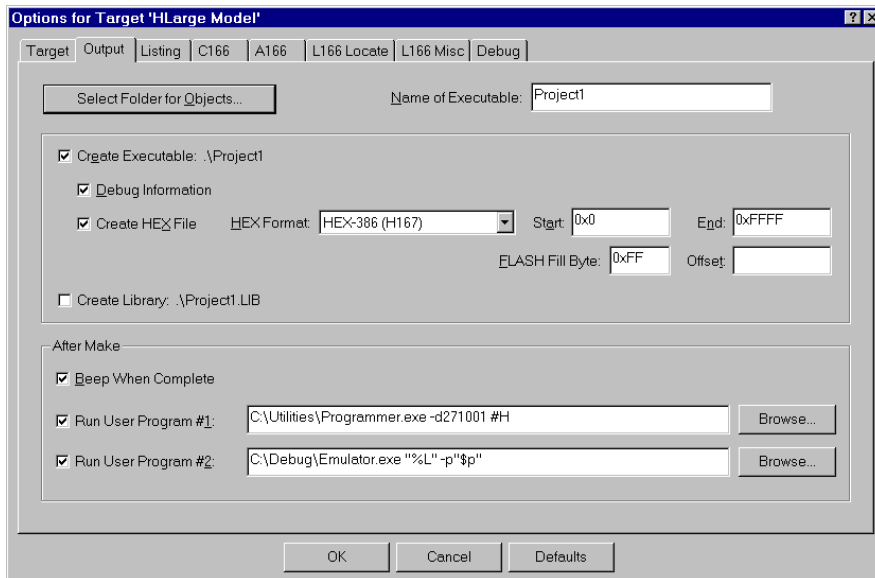
The **Options for Target – Output** dialog allows to enter up to two users programs that are started after a successful build process. Using a key sequence you may pass arguments from the  $\mu$  Vision2 project manager to these user programs. A key sequence is a combination of a **Key Code** and a **File Code**. The available **Key Codes** and **File Codes** are listed in the tables below:

Key Code	Specifies the path selected with the File Code
\$	folder name of the file specified in the file code (C:\MYPROJECT)
#	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)

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 inserted in the user program line
P	name of the current project file (PROJECT1.UV2)
X	$\mu$ Vision2 executable program file (C:\KEIL\UV2\UV2.EXE)
H	application HEX file (PROJECT1.H86)
L	linker output file, typical the executeable file for debugging (PROJECT1)

4



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.H86`. 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,  $\mu$ Vision2 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 $\mu$ Vision2 Device Database

The  $\mu$ Vision2 device database contains all 166 / ST10 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 **C166 (all Variants)** device supports all CPU's with no extended instruction set. The **C167 (all Variants)** is used for devices with extended instruction set. All new devices are based on the extended instruction set. Therefore most likely you will need to select this device. Specify the on-chip memory as External Memory in the **Options for Target – Target** dialog.
- You may enter a new CPU into the  $\mu$ Vision2 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 ...
<b>IRAM (range)</b>	Address location of the on-chip IRAM.
<b>XRAM (range)</b>	Address location of the on-chip XRAM.
<b>IROM (range)</b>	Address location of the on-chip (flash) ROM. The start address must be 0; the part is split automatically into two sections, if the size is more than 32KB. The range specifies the physical ROM size.
<b>ICAN (range)</b>	Address location of the on-chip CAN module.
<b>CLOCK (val)</b>	Default CPU clock used when you select the device.
<b>MOD167</b>	Use the extended instruction set.

Other **Option** variables specify CPU data books and  $\mu$ Vision2 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**.  $\mu$ Vision2 will call the LIB166 Library Manager instead of the L166 Linker/Locater. Since the code in the Library will be not linked and located, the entries in the **L166 Locate** and **L166 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 166/ST10 directives. Read the section “Data Threshold” on page 50 and check if you need to enter a **Target – Data Threshold** value in the options dialog.

The directive **NOFIXDPP** should be entered under **Options – C166 – Misc**, if the library is designed for a target application with one of the following configurations:

- More than 16KB ROM or 16KB ROM are set for **Target – Near Memory**. In this case the L166 **DPPUSE** directive will be applied. This directive requires that the C166 compiler made no default assumptions for the DPP registers.
- The User Stack is set to the memory class **SDATA** or **IDATA** as described under “Applications without external RAM Devices” on page 55. You may use the C166 directive **USERSTACKDPP3** instead of **NOFIXDPP**, but **NOFIXDPP** is more generic since no DPP register assumptions are made.

## 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 Sections 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 `0x128000`. This structure is defined in a source file named `ALMCTRL.C` and this module contains only the declaration for this structure.

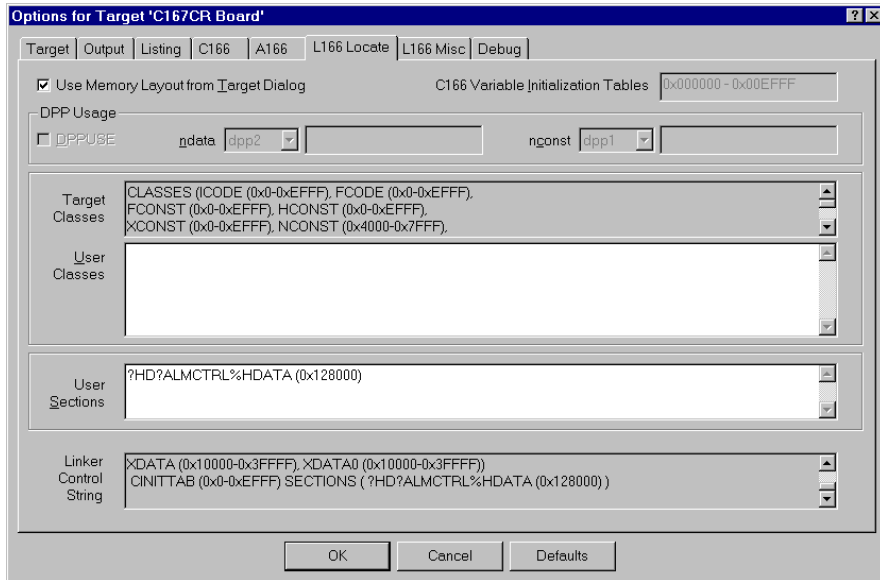
```

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

#pragma NOINIT          // disable zero initialization for alarm_control
struct alarm_st huge alarm_control;
:
:

```

The C166 compiler generates an object file for `ALMCTRL.C` and includes a section for variables in the **huge** memory area. The variable `alarm_control` is located in the section `?HD?ALMCTRL`.  $\mu$ Vision2 allows you to specify the base address of any section under **Options for Target – L166 Locate – Users Sections**.



In this example L166 will locate the section named `?HD?ALMCTRL` of the class `HDATA0` at address `0x128000` in the physical memory.

## User Classes

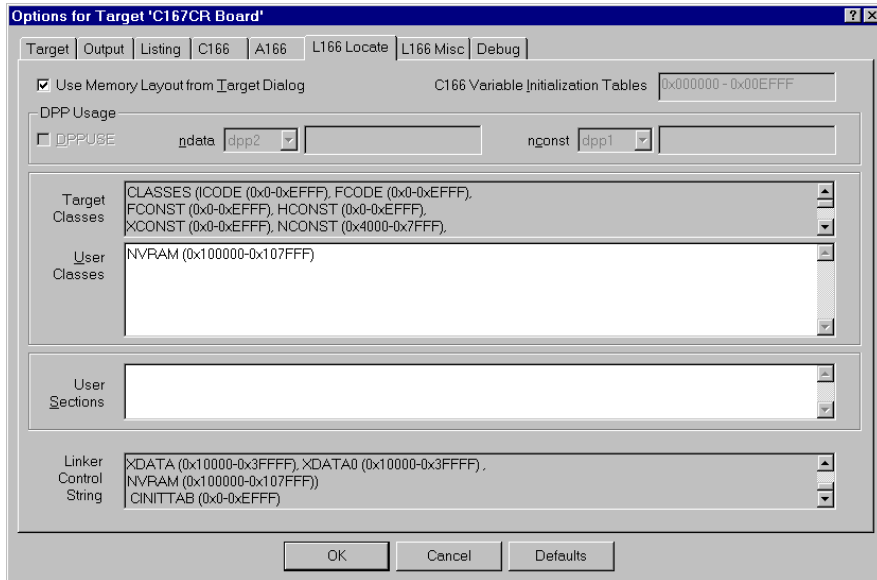
User classes are useful when you need to locate many variables from different source modules to special memory areas. In the following example we have a non-volatile RAM starting at address `0x100000`. You can use the C166 directive `RENAMECLASS` to rename the standard C166 `HDATA` memory class to `NVRAM` for several modules. With `#pragma NOINIT` the variable zero initialization is disabled. `µVision2` allows you to specify the base address of any section under **Options for Target – L166 Locate – Users Classes**.

### Source Module 1:

```
#pragma RENAMECLASS (HDATA=NVRAM)
:
:
#pragma NOINIT
    int huge value1;
static int huge value2;
:
```

### Source Module 2:

```
#pragma RENAMECLASS (HDATA=NVRAM)
:
:
#pragma NOINIT
static float huge value3;
    long huge value4;
:
```



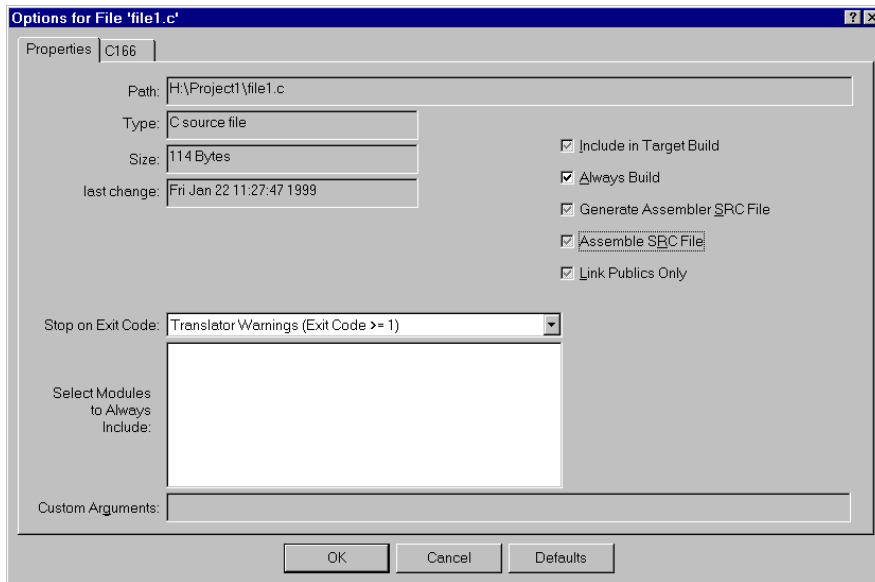
In this example L166 will locate the user class section named **NVRAM** in the physical memory address range 0x100000 - 0x107FFF.

## File and Group Specific Options – Properties Dialog

$\mu$ Vision2 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, $\mu$ Vision2 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 <b>__DATE__</b> and <b>__TIME__</b> macros that are used to stored version information in the application program.

Dialog Item	Description
Generate Assembler SRC File	Instructs the C166 compiler to generate an assembler source file from this C module. Typical this option is used when the C source file contains <b>#pragma asm / endasm</b> sections.
Assemble SRC File	Use this option together with the <b>Generate Assembler SRC File</b> to translate the assembler source code generated by C166 into an object file that can be linked to the application.
Link Publics Only	Instructs L166 to link 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, $\mu$ Vision2 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 64 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 64 for more information.



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 C166 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.

---

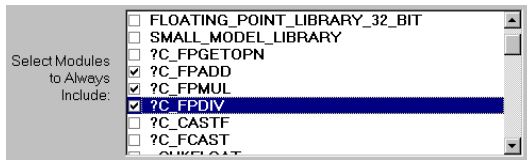
### NOTE

*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 C166 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

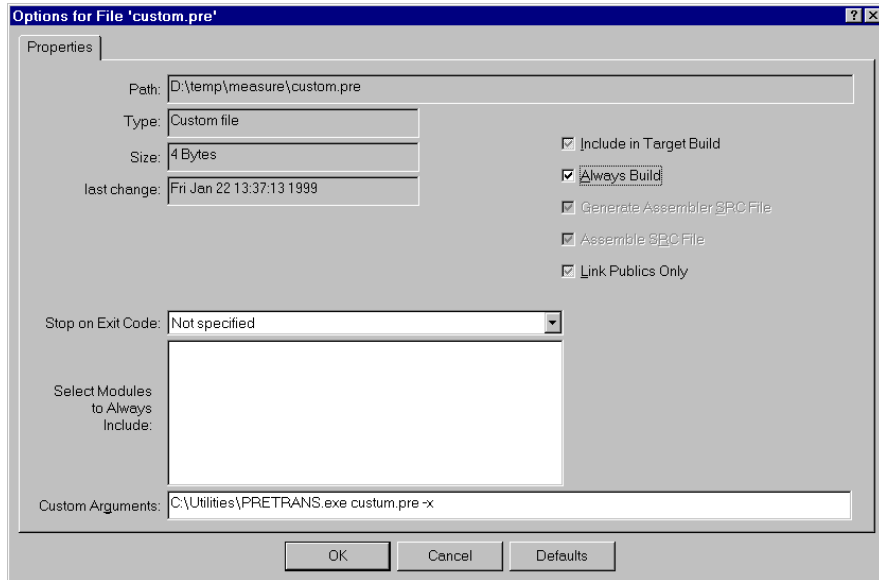
The Properties dialog page allows you to specify library modules that should be always included in a project. This is sometimes required when you generate a boot portion of an application that should contain generic routines that are used from program parts that are reloaded later. In this case add the 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**.



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,  $\mu$ Vision2 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 A166 or C166 to generate an object file that can be linked to your application.



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 C166 compiler and A166 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 166 / ST10 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 166** interface. With the Advanced GDI interface you may connect the environment directly to emulators, OCDS debugging systems or the Keil Monitor program. For more information refer to “Chapter 11. Using Monitor-166” on page 155.

### 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 166 / ST10 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 36 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,

$\mu$ Vision2 will load the application program and run the startup code. For information about the configuration of the  $\mu$ Vision2 debugger refer to page 74.  $\mu$ Vision2 saves the editor screen layout and restores the screen layout of the last debug session. If the program execution stops,  $\mu$ Vision2 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  $\mu$ Vision2 debug mode differs from the edit mode in the following aspects:

- The “Debug Menu and Debug Commands” described on page 17 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 lets you view your target program as mixed source and assembly program or just assembly code. In addition, a trace history of previously executed instructions can 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 CONF
-2  000233E6 76E30004 OR        DP3,#0x0400
196:  DP3 &= 0xF7FF; /* RESET PORT 3.11 DIRECTION CON
-1  000233EA 66E3FFF7 AND        DP3,#0xF7FF
197:  SOTIC = 0x80; /* SET TRANSMIT INTERRUPT FLAG
000233EE E6B68000 MOV        SOTIC,#0x0080
198:  SORIC = 0x00; /* DELETE RECEIVE INTERRUPT FLAG
000233F2 E6B70000 MOV        SORIC,#0x0000
199:  SOBG = 0x40; /* SET BAUDRATE TO 9600 BAUD
000233F6 E65A4000 MOV        SOBG,#0x0040
200:  SOCON = 0x8011; /* SET SERIAL MODE
201:  #endif
202:
203:  /* setup the timer 0 interrupt */
000233FA E6D81180 MOV        SOCON,#0x8011
204:  TOREL = PERIOD; /* set reload value */
000233FE E62A3CF6 MOV        TOREL,#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.

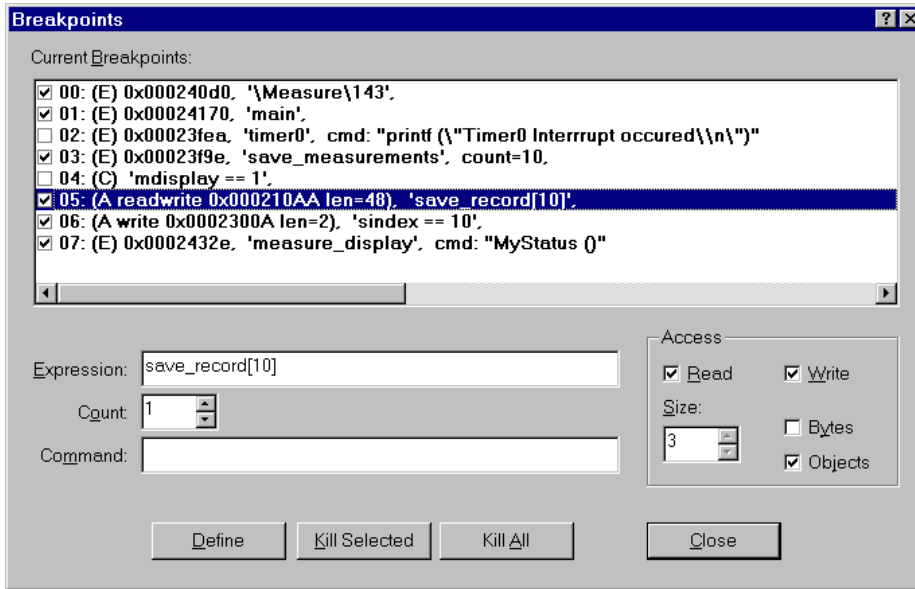


## Breakpoints

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.



## 5

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,  $\mu$ Vision2 executes the command and resumes executing your target program. The command you specify here may be a  $\mu$ Vision2 debug or signal function. To halt program execution in a  $\mu$ Vision2 function, set the `_break_` system variable. For more information refer to “System Variables” on page 85.

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 disabled 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 10<sup>th</sup> time.

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

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

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

**Access Break (A)** that halts program execution when a 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  $\mu$ Vision2 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

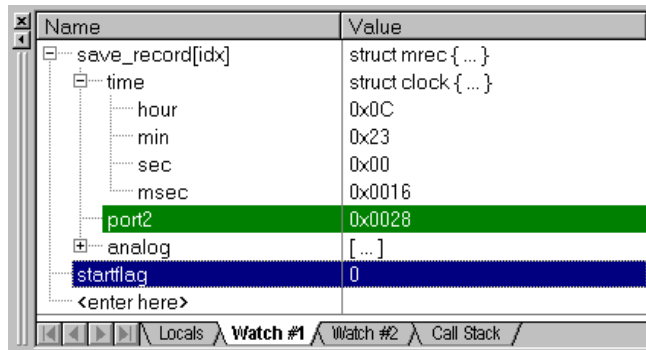
$\mu$ Vision2 lets execute your application program in several different ways:

- With the **Debug Toolbar** buttons and the “Debug Menu and Debug Commands” as described on page 17.
- 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.

# 5

## 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.



The **Locals** page shows all local function variables of the current function. The **Watch** pages display user-specified 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**.  $\mu$ Vision2 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.



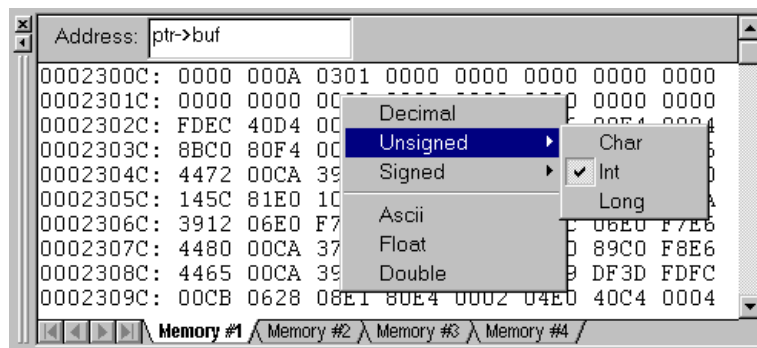
## CPU Registers

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.



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

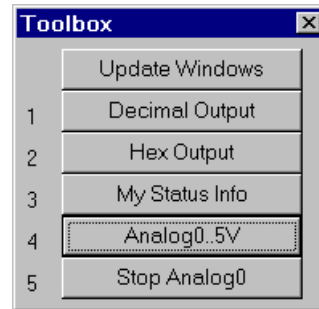
The **Toolbox** contains user-configurable buttons. Click on a **Toolbox** button to execute the associated command. Toolbox buttons may be executed at any time, even while running the test program.

Toolbox buttons are defined in 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  $\mu$  Vision2 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

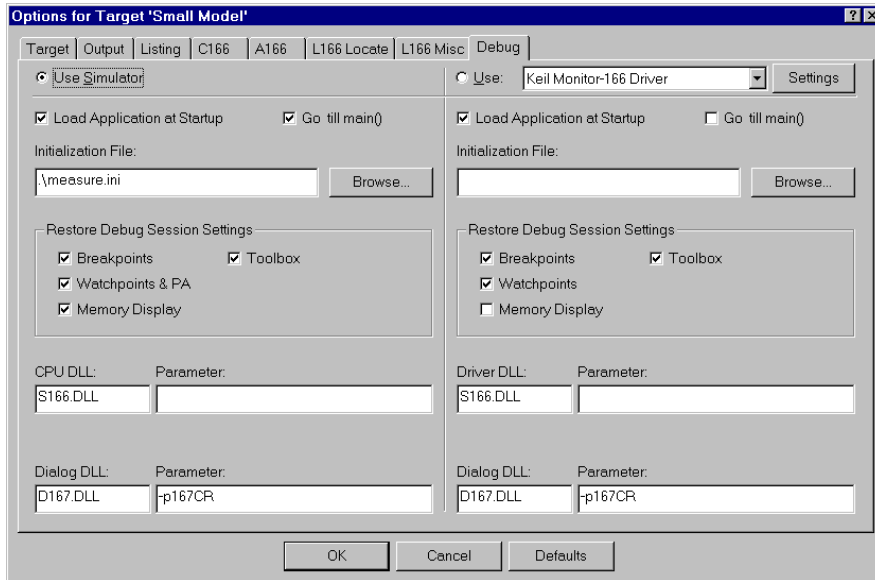
The **Update Windows** button in the **Toolbox** is created automatically and cannot be removed. The **Update Windows** button updates several debug windows during program execution.





## Set Debug Options

The dialog **Options for Target - Debug** configures the  $\mu$ Vision2 debugger.



5

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

Dialog Item	Description
Use Simulator	Select the $\mu$ Vision2 Simulator as Debug engine.
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 $\mu$ Vision2 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 $\mu$ Vision2 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.

Dialog Item	Description
CPU DLL Driver DLL Parameter	Configures the internal $\mu$ Vision2 debug DLLs. The settings are taken from the device database. Please do not modify the DLL or DLL parameters.



## Serial Window

$\mu$ Vision2 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.

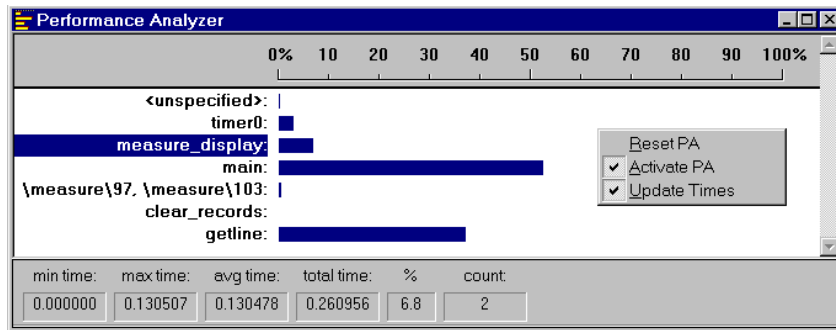
```

Serial #1
+***** 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        |         | measurement values
| Time     | T hh:mm:ss|         |
| Interval | I mm:ss.ttt |         |
| Clear    | C        |         | records
| 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  $\mu$ Vision2 Performance Analyzer displays the execution time recorded for functions and address ranges you specify.



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.

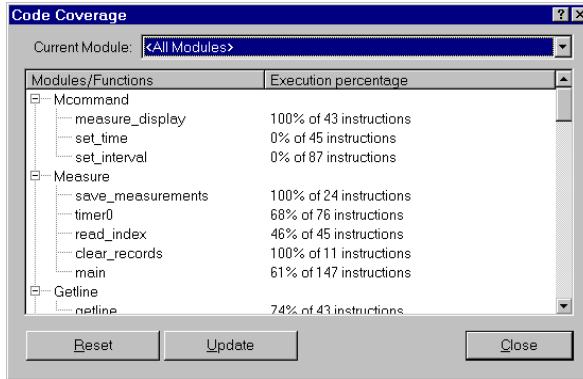
Label	Description
<b>min time</b>	The minimum time spent in the selected address range or function.
<b>max time</b>	The maximum time spent in the selected address range or function.
<b>avg time</b>	The average time spent in the selected address range or function.
<b>total time</b>	The total time spent in the selected address range or function.
<b>%</b>	The percent of the total time spent in the selected address range or function.
<b>count</b>	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  $\mu$ Vision2 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 marked 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.



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

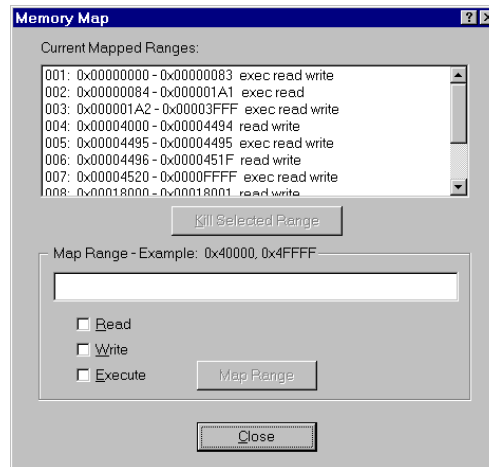
## Memory Map

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,  $\mu$ Vision2 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.

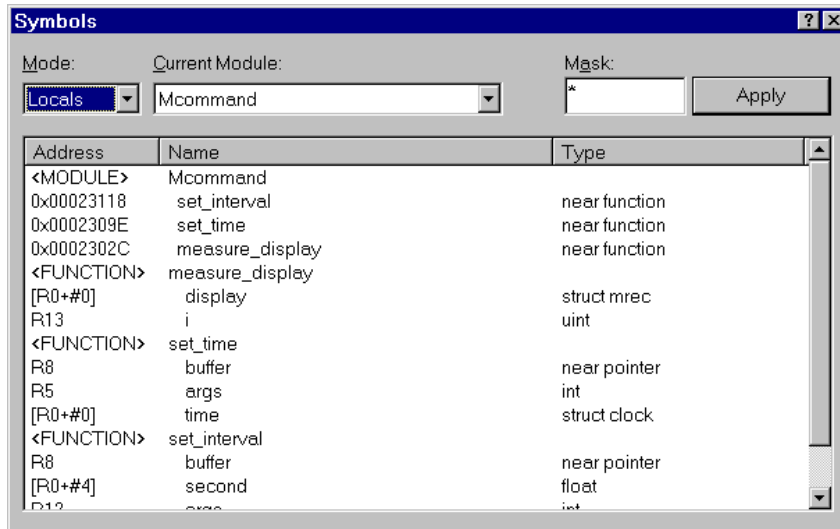
The dialog opens via the menu **Debug – Memory Map...**

As your target program runs,  $\mu$ Vision2 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.



## 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.



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: <ul style="list-style-type: none"> <li># matches a digit (0 – 9)</li> <li>\$ matches any character</li> <li>* matches zero or more characters.</li> </ul>
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.
<u>a</u> \$#*	... with an underline, followed by the letter <b>a</b> , followed by any character, followed by a digit, ending with zero or more characters. For example, <b><u>a</u>10value</b> .

Mask	Matches symbol names ...
<u>_</u> *ABC	... with an underline, followed by zero or more characters, followed by <b>ABC</b> .

## Debug Commands

You interact with the  $\mu$ Vision2 debugger by entering commands from the keyboard in the **Command** page of the **Output Window**. You can enter nearly all debug commands. In the following all available  $\mu$ Vision2 debug commands are listed in categories.

### NOTE

Use the underlined characters in the command names to enter commands. For example, the **WATCHSET** command must be entered as **WS**.

## Memory Commands

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

Command	Description
<u>A</u> SM	Assembles in-line code.
<u>D</u> EFINE	Defines typed symbols that you may use with $\mu$ Vision2 debug functions.
<u>D</u> ISPLAY	Display the contents of memory.
<u>E</u> NTER	Enters values into a specified memory area.
<u>E</u> VALUATE	Evaluates an expression and outputs the results.
<u>M</u> AP	Specifies access parameters for memory areas.
<u>U</u> NASSEMBLE	Disassembles program memory.
<u>W</u> ATCHSET	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
<u>E</u> sc	Stops program execution.
<u>G</u> O	Starts program execution.
<u>P</u> STEP	Steps over instructions but does not step into procedures or functions.
<u>O</u> STEP	Steps out of the current function.
<u>I</u> STEP	Steps over instructions and into functions.

## Breakpoint Commands

$\mu$ Vision2 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
<b><u>BREAKDISABLE</u></b>	Disables one or more breakpoints.
<b><u>BREAKENABLE</u></b>	Enables one or more breakpoints.
<b><u>BREAKKILL</u></b>	Removes one or more breakpoints from the breakpoint list.
<b><u>BREAKLIST</u></b>	Lists the current breakpoints.
<b><u>BREAKSET</u></b>	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
<b><u>ASSIGN</u></b>	Assigns input and output sources for the Serial window.
<b><u>COVERAGE</u></b>	List code coverage statistics.
<b><u>DEFINE BUTTON</u></b>	Creates a Toolbox button.
<b><u>DIR</u></b>	Generates a directory of symbol names.
<b><u>EXIT</u></b>	Exits the $\mu$ Vision2 debug mode.
<b><u>INCLUDE</u></b>	Reads and executes the commands in a command file.
<b><u>KILL</u></b>	Deletes $\mu$ Vision2 debug functions and Toolbox buttons.
<b><u>LOAD</u></b>	Loads CPU drivers, object modules, and HEX files.
<b><u>LOG</u></b>	Creates log files, queries log status, and closes log files for the Debug window.
<b><u>MODE</u></b>	Sets the baud rate, parity, and number of stop bits for PC COM ports.
<b><u>PerformanceAnalyze</u></b>	Setup the performance analyzer or list PA information.
<b><u>RESET</u></b>	Resets CPU, memory map assignments, Performance Analyzer or predefined variables.
<b><u>SAVE</u></b>	Saves a memory range in an Intel HEX386 file.
<b><u>SCOPE</u></b>	Displays address assignments of modules and functions of a target program.
<b><u>SET</u></b>	Sets the string value for predefined variable.
<b><u>SETMODULE</u></b>	Assigns a source file to a module.
<b><u>SIGNAL</u></b>	Displays signal function status and removes active signal functions.
<b><u>SLOG</u></b>	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
<b>MDH</b>	Display the <b>MDH</b> register.
<b>R7 = 12</b>	Assign the value 12 to register <b>R7</b> .
<b>time.hour</b>	Displays the member <b>hour</b> of the <b>time</b> structure.
<b>time.hour++</b>	Increments the member <b>hour</b> of the <b>time</b> structure.
<b>index = 0</b>	Assigns the value 0 to <b>index</b> .

## 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.

# 5

## Components of an Expression

An expression may consist of any of the following components.

Component	Description
<b>Bit Addresses</b>	Bit addresses reference bit-addressable data memory.
<b>Constants</b>	Constants are fixed numeric values or character strings.
<b>Line Numbers</b>	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.
<b>Operators</b>	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.
<b>Program Variables (Symbols)</b>	Program variables are those variables in your target program. They are often called symbols or symbolic names.
<b>System Variables</b>	System variables alter or affect the way $\mu$ Vision2 operates.
<b>Type Specifications</b>	Type specifications let you specify the data type of an expression or subexpression.

## Constants

The  $\mu$ Vision2 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
<b>Binary:</b>	None	<b>Y</b> or <b>y</b>	<b>11111111Y</b>
<b>Decimal:</b>	None	<b>T</b> or none	<b>1234T</b> or <b>1234</b>
<b>Hexadecimal:</b>	<b>0x</b> or <b>0X</b>	<b>H</b> or <b>h</b>	<b>1234H</b> or <b>0x1234</b>
<b>Octal:</b>	None	<b>Q</b> , <b>q</b> , <b>O</b> , or <b>o</b>	<b>777q</b> or <b>777Q</b> or <b>777o</b>

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<sup>[+|-]</sup> number*

*number . number [e<sup>[+|-]</sup> 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  $\mu$ Vision2 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	Sequence	Description
\\	Backslash character ("").	\n	Newline.
\"	Double quote.	\r	Carriage return.
'	Single quote.	\t	Tab.
\a	Alert, bell.	\0nn	Octal constant.
\b	Backspace.	\Xnnn	HEX constant.
\f	Form feed.		

## 5

## String Constants

The rules of the C programming language for string constants also apply to  $\mu$ Vision2. 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
<b>\$</b>	<b>unsigned long</b>	represents the program counter. You may use \$ to display and change the program counter. For example, \$ = 0x4000 sets the program counter to address 0x4000.
<b>_break_</b>	<b>unsigned int</b>	lets you stop executing the target program. When you set <b>_break_</b> to a non-zero value, $\mu$ Vision2 halts target program execution. You may use this variable in user and signal functions to halt program execution. Refer to “Chapter 6. $\mu$ Vision2 Debug Functions” on page 97 for more information.
<b>_traps_</b>	<b>unsigned int</b>	when you set <b>_traps_</b> to a non-zero value, $\mu$ Vision2 display messages for the 166 hardware traps: Undefined Opcode, Protected Instruction Fault, Illegal Word Operand Access, Illegal Instruction Access, Stack Underflow and Stack Overflow.
<b>states</b>	<b>unsigned long</b>	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. <b>NOTE:</b> <i>states</i> is a read-only variable.
<b>itrace</b>	<b>unsigned int</b>	indicates whether or not trace recording is performed during target program execution. When <b>itrace</b> is 0, no trace recording is performed. When <b>itrace</b> has a non-zero value, trace information is recorded. Refer to page 68 for more information.
<b>radix</b>	<b>unsigned int</b>	determines the base used for numeric values displayed. <b>radix</b> may be 10 or 16. The default setting is 16 for HEX output.

## On-chip Peripheral Symbols

$\mu$ Vision2 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)

$\mu$ Vision2 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
<b>AINx</b>	An analog input pin on the chip. Your target program may read values you write to <b>AINx</b> VTREGs.
<b>PORTx</b>	A group of I/O pins for a port on the chip. For example, <b>PORT2</b> refers to all 8 or 16 pins of P2. These registers allow you to simulate port I/O.
<b>SxIN</b>	The input buffer of serial interface <b>x</b> . You may write 8-bit or 9-bit values to <b>SxIN</b> . These are read by your target program. You may read <b>SxIN</b> 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.
<b>SxOUT</b>	The output buffer of serial interface <b>x</b> . $\mu$ Vision2 copies 8-bit or 9-bit values (as programmed) to the <b>SxOUT</b> VTREG.
<b>SxTIME</b>	Defines the baudrate timing of the serial interface <b>x</b> . When <b>SxTIME</b> is 1, $\mu$ Vision2 simulates the timing of the serial interface using the programmed baudrate. When <b>SxTIME</b> is 0 (the default value), the programmed baudrate timing is ignored and serial transmission time is instantaneous.
<b>CLOCK</b>	The clock frequency of the simulated CPU as defined in the <b>Options – Target</b> dialog.

5

### 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 C167 CPU the following VTREG symbols for the on-chip peripheral registers are available:

CPU-pin Symbol	Description
<b>AIN0</b>	Analog input line AIN0 (floating-point value)
<b>AIN1</b>	Analog input line AIN1 (floating-point value)
<b>AIN2</b>	Analog input line AIN2 (floating-point value)

CPU-pin Symbol	Description										
<b>AIN3</b>	Analog input line AIN3 (floating-point value)										
<b>AIN4</b>	Analog input line AIN4 (floating-point value)										
<b>AIN5</b>	Analog input line AIN5 (floating-point value)										
<b>AIN6</b>	Analog input line AIN6 (floating-point value)										
<b>AIN7</b>	Analog input line AIN7 (floating-point value)										
<b>AIN8</b>	Analog input line AIN8 (floating-point value)										
<b>AIN9</b>	Analog input line AIN9 (floating-point value)										
<b>AIN10</b>	Analog input line AIN10 (floating-point value)										
<b>AIN11</b>	Analog input line AIN11 (floating-point value)										
<b>AIN12</b>	Analog input line AIN12 (floating-point value)										
<b>AIN13</b>	Analog input line AIN13 (floating-point value)										
<b>AIN14</b>	Analog input line AIN14 (floating-point value)										
<b>AIN15</b>	Analog input line AIN15 (floating-point value)										
<b>EA</b>	Status of the EA pin (1 bit). This configuration pin is necessary for calculating the execution time of a program. You must invoke the <b>RESET</b> command after changing the value of <b>EA</b> .										
<b>EBC</b>	External Bus Configuration after reset (2 bits). <b>EBC</b> may have one of the following values: <table border="1" data-bbox="578 811 1230 930"> <thead> <tr> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><b>0</b></td> <td>8-bit data bus, non-multiplexed</td> </tr> <tr> <td><b>1</b></td> <td>8-bit data bus, multiplexed</td> </tr> <tr> <td><b>2</b></td> <td>16-bit data bus, non-multiplexed</td> </tr> <tr> <td><b>3</b></td> <td>16-bit data bus, multiplexed</td> </tr> </tbody> </table> <p>Bus configuration pins are necessary for calculating the execution time of a program. Bit 0 of <b>EBC</b> represents EBC0. Bit 1 of <b>EBC</b> represents EBC1. When <b>EBC</b> is 3, EBC0 and EBC1 are both set. You must invoke the <b>RESET</b> command after changing the value of <b>EBC</b>.</p>	Value	Description	<b>0</b>	8-bit data bus, non-multiplexed	<b>1</b>	8-bit data bus, multiplexed	<b>2</b>	16-bit data bus, non-multiplexed	<b>3</b>	16-bit data bus, multiplexed
Value	Description										
<b>0</b>	8-bit data bus, non-multiplexed										
<b>1</b>	8-bit data bus, multiplexed										
<b>2</b>	16-bit data bus, non-multiplexed										
<b>3</b>	16-bit data bus, multiplexed										
<b>PORT0H</b>	Digital I/O lines of PORT 0H (8-bit)										
<b>PORT0L</b>	Digital I/O lines of PORT 0L (8-bit)										
<b>PORT1H</b>	Digital I/O lines of PORT 1H (8-bit)										
<b>PORT1L</b>	Digital I/O lines of PORT 1L (8-bit)										
<b>PORT2</b>	Digital I/O lines of PORT 2 (16-bit)										
<b>PORT3</b>	Digital I/O lines of PORT 3 (16-bit)										
<b>PORT4</b>	Digital I/O lines of PORT 4 (8-bit)										
<b>PORT5</b>	Digital/analog Input Lines of PORT 5 (16-bit)										
<b>PORT6</b>	Digital I/O lines of PORT 6 (8-bit)										
<b>PORT7</b>	Digital I/O lines of PORT 7 (8-bit)										
<b>PORT8</b>	Digital I/O lines of PORT 8 (8-bit)										
<b>S0IN</b>	Serial input for SERIAL CHANNEL 0 (9 bits)										
<b>S0OUT</b>	Serial output for SERIAL CHANNEL 0 (9 bits)										
<b>S0TIME</b>	Serial timing enable for SERIAL CHANNEL 0										
<b>S1IN</b>	Serial input for SERIAL CHANNEL 1 (9 bits)										
<b>S1OUT</b>	Serial output for SERIAL CHANNEL 1 (9 bits)										

CPU-pin Symbol	Description
<b>S1TIME</b>	Serial timing enable for SERIAL CHANNEL 1
<b>CLOCK</b>	Clock 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

$\mu$ Vision2 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  $\mu$ Vision2, 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) {
        PORT2 |= 1;          /* repeat forever */
        twatch ((CLOCK / 2) / 2000); /* set P1.2 */
        PORT2 &= ~1;        /* delay for .0005 secs */
        twatch ((XCLOCK / 2) / 2000); /* clear P1.2 */
        PORT2 &= ~1;        /* delay for .0005 secs */
    }
}

```

The following command starts this signal function:

```
one_thou_hz ()
```

Refer to “Chapter 6.  $\mu$ Vision2 Debug Functions” on page 97 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  $\mu$ Vision2 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) {
        printf ("LED is ON\n"); }
    else {
        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 69 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 is instantaneous (**STIME** = 0) or the serial port timing is relative to the specified baudrate (**SxTIME** = 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) {
        twatch (CLOCK);
        S0IN = 'A';
        twatch (CLOCK / 900);
        S0IN = 'B';
        twatch (CLOCK / 900);
        S0IN = 'C';
    }
}

```

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**.*

---



## Symbol Classification

$\mu$ Vision2 recognizes two classes of symbols:

**Program Symbols** are defined in the target program.

**Reserved Words** are predefined in  $\mu$ Vision2. Debug commands & options, data type names, register names, system variables, CPU symbols, and VTREGs are all reserved words.

## Literal Symbols

Often you may find that your program symbols duplicate reserved words. When this occurs, you must literalize your program symbol using the back quote character (`) to differentiate them from reserved words.

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

### Accessing the R5 Register

```
>R5 = 121
```

### Accessing the R5 Variable

```
>`R5 = 212
```

Normally,  $\mu$ Vision2 searches for reserved words then for target program symbols. When you literalize a symbol,  $\mu$ Vision2 only searches for target program symbols.

## Fully Qualified Symbols

Symbols may be entered using the only name of the variable or function they reference. Symbols may also 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 may include any 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.

You may combine names as shown in the following table:

Symbol Components	Full Qualified Symbol Name addresses ...
<code>\ModuleName\LineNumber</code>	... line number <i>LineNumber</i> in <i>ModuleName</i> .
<code>\ModuleName\FunctionName</code>	... <i>FunctionName</i> function in <i>ModuleName</i> .
<code>\ModuleName\SymbolName</code>	... global symbol <i>SymbolName</i> in <i>ModuleName</i> .
<code>\ModuleName\FunctionName\SymbolName</code>	... local symbol <i>SymbolName</i> in the <i>FunctionName</i> function in <i>ModuleName</i> .

Examples of fully qualified symbol names:

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

## Non-Qualified Symbols

When you enter a fully qualified symbol name,  $\mu$ Vision2 determines if the symbol exists and reports an error if it does not. For symbols that are not fully qualified,  $\mu$ Vision2 searches a number of tables until a matching symbol name is found. This search works as follows:

1. **Local Variables in the Current Function** in the target program. The current function is determined by the value of the program counter.
2. **Global or 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. This includes file-scope or static variables.
3. **Symbols Created with the  $\mu$ Vision2 DEFINE Command.** These symbols are used for debugging and are not a part of the target program.
4. **System Variables** provide a way to monitor and change debugger characteristics. They are not a part of the target program. Refer to “System Variables” on page 85 for more information. If a global variable in your target

program shares the same name as a system variable, you may access the global variable using a literal symbol name. Refer to “Literal Symbols” on page 91 for more information.

5. **Global or Public Symbols** of your target program. SFRs defined by  $\mu$ Vision2 are considered to be public symbols and are also searched.
6. **CPU Driver Symbols (VTREGs)** defined by the CPU driver. Refer to “CPU Pin Registers (VTREGs)” on page 86 for a description of VTREG symbols.

---

### NOTE

*The search order for symbols changes when creating user or signal functions.  $\mu$ Vision2 first searches the table of symbols defined in the user or signal function. Then, the above list is searched. Refer to “Chapter 6.  $\mu$ Vision2 Debug Functions” on page 97 for more information about user and signal functions.*

---

## 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,  $\mu$ Vision2 lets you use in an expression. The syntax for a line number is shown in the following table.

Line Number Symbol	Code Address ...
<code>\LineNumber</code>	... for line number <i>LineNumber</i> in the current module.
<code>\ModuleName\LineNumber</code>	... for line number <i>LineNumber</i> in <i>ModuleName</i> .

### 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 97 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, mem type = C: */
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

```

## 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

```

## Chapter 6. $\mu$ Vision2 Debug Functions

This chapter discusses a powerful aspect of the  $\mu$ Vision2: debug functions. You may use functions to extend the capabilities of the  $\mu$ Vision2 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 not confuse  $\mu$ Vision2 debug functions with functions of your target program.  $\mu$ Vision2 debug functions aids you in debugging of your application and are entered or with the **Function Editor** or on  $\mu$ Vision2 command level.*

---

$\mu$ Vision2 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  $\mu$ Vision2 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 110.

### Creating Functions

$\mu$ Vision2 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  $\mu$ Vision2 editor and allows you to enter and compile debug functions.

```

Function Editor - measure.ini
Open New... Save Save As... Compile
Compile Errors:
/*-----*/
/* 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 $\mu$ Vision2 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 $\mu$ Vision2 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  $\mu$ Vision2 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,  $\mu$ Vision2 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  $\mu$ Vision2 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  $\mu$ Vision2 debugger responds by printing the text “Hello World” in the Command page of the Output Window.

## Function Classes

$\mu$ Vision2 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  $\mu$ Vision2 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

$\mu$ Vision2 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  $\mu$ Vision2 debug functions.

Return	Name	Parameter	Description
void	<b>exec</b>	("command_string")	Execute Debug Command
double	<b>getdbl</b>	("prompt_string")	Ask the user for a double number
int	<b>getint</b>	("prompt_string")	Ask the user for a int number
long	<b>getlong</b>	("prompt_string")	Ask the user for a long number
void	<b>memset</b>	(start_addr, value, len)	fill memory with constant value
void	<b>printf</b>	("string", ...)	works like the ANSI C printf function
int	<b>rand</b>	(int seed)	return a random number in the range -32768 to +32767
void	<b>twatch</b>	(ulong states)	Delay execution of signal function for specified number of CPU states
uchar	<b>_RBYTE</b>	(address)	Read <b>char</b> on specified memory address
uint	<b>_RWORD</b>	(address)	Read <b>int</b> on specified memory address
ulong	<b>_RDWORD</b>	(address)	Read <b>long</b> on specified memory address
float	<b>_RFLOAT</b>	(address)	Read <b>float</b> on specified memory address
double	<b>_RDOUBLE</b>	(address)	Read <b>double</b> on specified memory address
void	<b>_WBYTE</b>	(address, uchar val)	Write <b>char</b> on specified memory address
void	<b>_WORD</b>	(address, uint val)	Write <b>int</b> on specified memory address
void	<b>_WDWORD</b>	(address, ulong val)	Write <b>long</b> on specified memory address
void	<b>_WFLOAT</b>	(address, float val)	Write <b>float</b> on specified memory address
void	<b>_WDOUBLE</b>	(address, double val)	Write <b>double</b> on specified memory address

The predefined functions are described below.

### void **exec** ("command\_string")

The **exec** function lets you invoke  $\mu$ Vision2 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 **printf** 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*.  $\mu$ Vision2 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.

**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  $\mu$ Vision2 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

*$\mu$ Vision2 provides a number of system variables you may use in your user functions. Refer to “**System Variables**” on page 85 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 example shows a user function that displays the contents of the registers R0 through R7. For more information about “Creating Functions” refer to page 97.

```
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  
-----
```

## Restrictions

$\mu$ Vision2 checks user functions to ensure they return values that correspond 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  $\mu$ Vision2 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  $\mu$ Vision2 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  $\mu$ Vision2 simulates your target program. Therefore, a signal function must call the **twatch** function at some point to delay and let  $\mu$ Vision2 run your target program.  $\mu$ Vision2 reports an error for signal functions that never call **twatch**.

---

### NOTE

*$\mu$ Vision2 provides a number of system variables you may use in your signal functions. Refer to “System Variables” on page 85 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 97.

```
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 an 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,  $\mu$ Vision2 may enter an infinite loop.

## Managing Signal Functions

$\mu$ Vision2 maintains a queue for active signal functions. A signal function may either be 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,  $\mu$ Vision2 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 from 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 displays the current state of the **analog0**:

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

$\mu$ Vision2 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  $\mu$ Vision2 debug user and signal functions.

$\mu$ Vision2 does not differentiate between uppercase and lowercase. The names of objects and control statements may be written in either uppercase or lowercase.

$\mu$ Vision2 has no preprocessor. Preprocessor directives like **#define**, **#include**, and **#ifdef** are not supported.

$\mu$ Vision2 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  $\mu$ Vision2, variables may not be initialized when they are declared. Explicit assignment statements must be used to initialize variables.

$\mu$ Vision2 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.

$\mu$ Vision2 functions may only return scalar variable types. Pointers and structures may not be returned.

$\mu$ Vision2 functions cannot be called recursively. During function execution,  $\mu$ Vision2 recognizes recursive calls and aborts function execution if one is detected.

$\mu$ Vision2 functions may only be invoked directly using the function name. Indirect function calls via pointers are not supported.

$\mu$ Vision2 supports the new ANSI style of function declaration for functions 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 */
{
  /* ... */
}
```

## 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\C166\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
<b>BADCODE</b>	Program with syntax errors and warnings. You may use the $\mu$ Vision2 editor to correct these.
<b>CSAMPLE</b>	Simple addition and subtraction calculator that shows how to build a multi- module project with $\mu$ Vision2.
<b>DHRY</b>	Dhrystone benchmark. Calculates the dhrystones factor for the target CPU.
<b>HELLO</b>	Hello World program. Try this first when you begin using $\mu$ Vision2. It prints Hello World on the serial interface and helps you confirm that the development tools work correctly. Refer to “HELLO: Your First 166 C Program” on page 112 for more information about this sample program.
<b>MEASURE</b>	Data acquisition system that collects analog and digital signals. Refer to “MEASURE: A Remote Measurement System” on page 117 for more information about this sample program.
<b>RTX_EX1</b>	Demonstrates round-robin multitasking using RTX-166 Tiny.
<b>RTX_EX2</b>	Demonstrates an RTX-166 Tiny application that uses signals.
<b>SIEVE</b>	Benchmark that calculates prime numbers.
<b>TRAFFIC</b>	Shows how to control a traffic light using the RTX-166 Tiny real-time executive.
<b>WHETS</b>	Benchmark program that calculates the whetstones factor for the target CPU.

To begin using one of the sample projects, use the  $\mu$  Vision2 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 C51 Program
- MEASURE: A Remote Measurement System

## HELLO: Your First 166 C Program

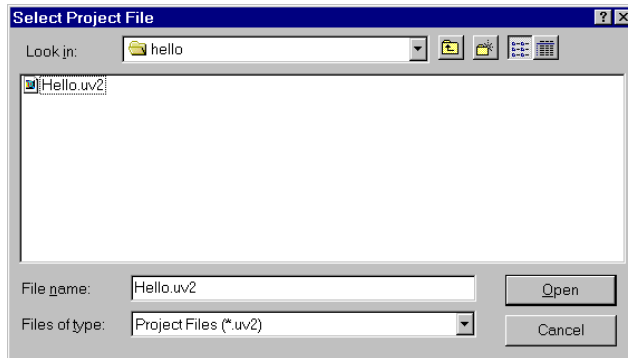
The HELLO sample program is located in `C:\KEIL\C166\EXAMPLES\HELLO\`. HELLO does nothing more than print the text “Hello World” to the serial port. The entire program is contained in a single source file `HELLO.C`.

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  $\mu$  Vision for Windows using the provided project file.

The hardware for HELLO is based on the standard C167 CPU. The only on-chip peripheral used is the serial port. You do not actually need a target CPU because  $\mu$  Vision2 lets you simulate the hardware required for this program.

### HELLO Project File

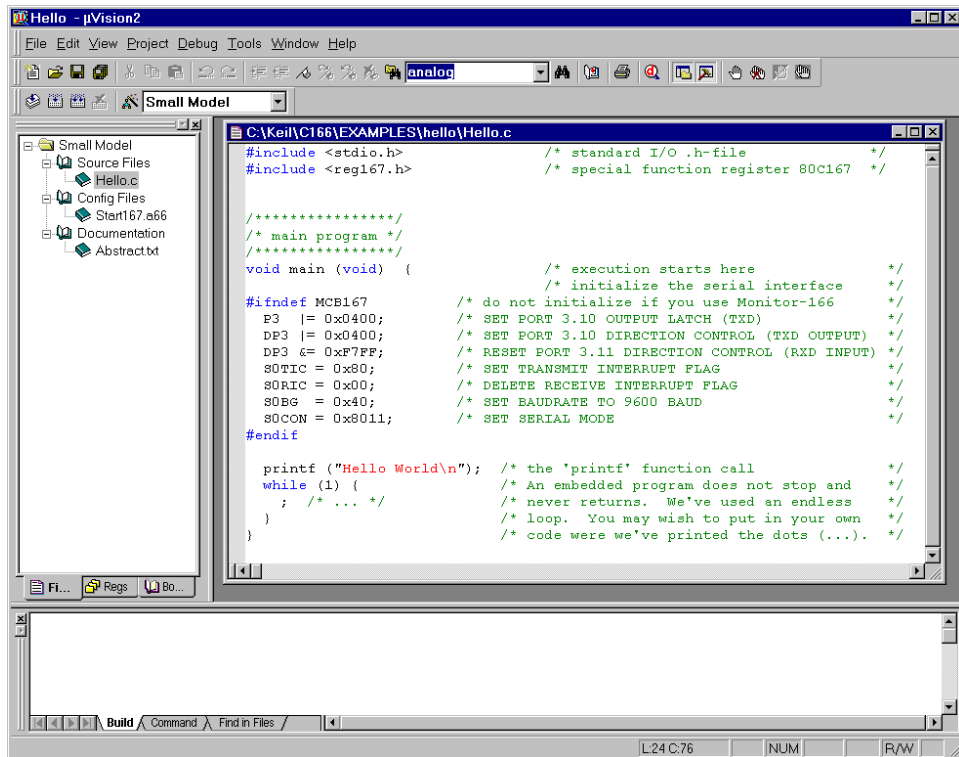
In  $\mu$  Vision, applications are maintained in a project file. A project file has been created for HELLO. To load this project, select **Open Project** from the **Project** menu and open `HELLO.UV2` from the folder `...\C166\EXAMPLES\HELLO`.



## 7

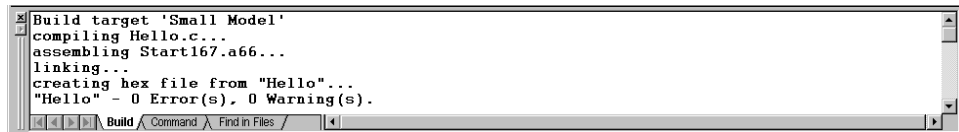
### Editing HELLO.C

You can now edit `HELLO.C`. Double click on `HELLO.C` in the Files page of the Project Window.  $\mu$  Vision2 loads and displays the contents of `HELLO.C` in an editor window.



## 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.  $\mu$ Vision2 begins to translate and link the source files and creates an absolute object module that you can load into the  $\mu$ Vision2 debugger for testing. The status of the build process is listed in the **Build** page of the **Output Window**.

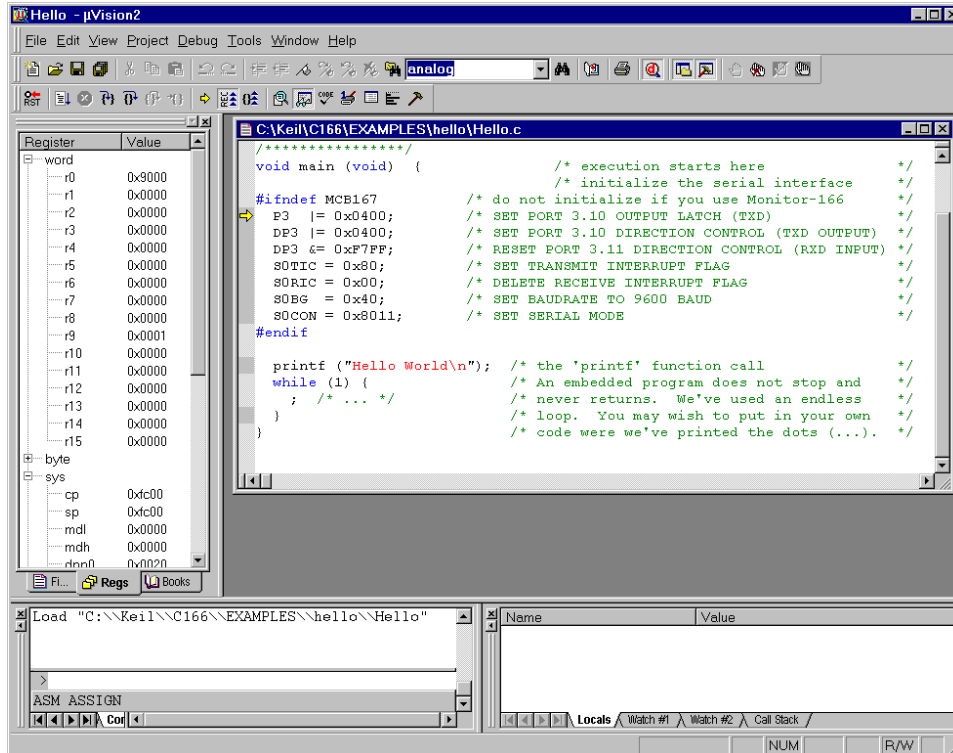


### **NOTE**

*You should encounter no errors when you use  $\mu$ Vision2 with the provided sample projects.*

## Testing HELLO

Once the HELLO program is compiled and linked, you can test it with the  $\mu$ Vision2 debugger. In  $\mu$ Vision2, use the **Start/Stop Debug Session** command from the **Debug** menu or toolbar.  $\mu$ Vision2 initializes the debugger and starts program execution till the main function. The following screen displays.



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.



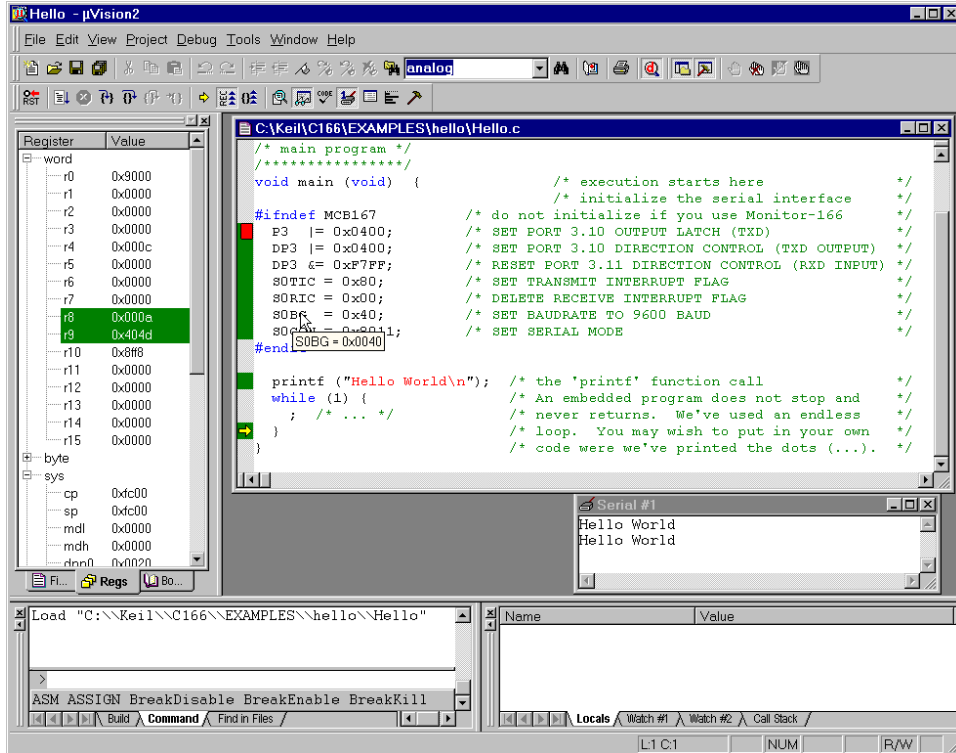
**Run HELLO** with the **Go** command from the **Debug** menu or toolbar. The HELLO program executes and displays the text “Hello World” in the serial window. After HELLO outputs “Hello World,” it begins executing an endless loop.



**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  $\mu$ Vision2 will show the following output:



## 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**,  $\mu$ Vision2 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

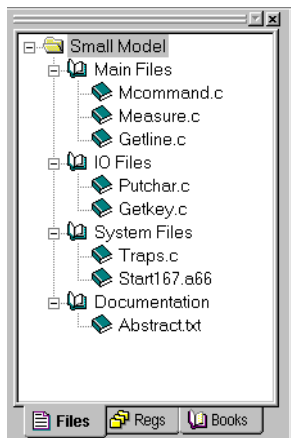
The MEASURE sample program is located in the \C166\EXAMPLES\MEASURE\ folder. MEASURE runs a remote measurement system that collects analog and digital data like a data acquisition systems found in a weather stations and process control applications. MEASURE is composed of three source files: GETLINE.C, MCOMMAND.C, and MEASURE.C.

This implementation records data from one 16-bit digital port 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

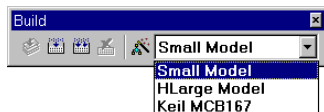
The hardware for MEASURE is based on the C167 CPU. This microcontroller provides analog and digital input capability. Port 2 is used for the digital inputs and AN0 through AN3 are used for the analog inputs. You do not actually need a target CPU because  $\mu$  Vision2 lets you simulate all the hardware 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\C166\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 C167 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.  $\mu$ Vision2 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 44.



## 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 C167 CPU. If you do not have target hardware, you can use  $\mu$ Vision2 to simulate the hardware. You can also use the serial window in  $\mu$ Vision2 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  $\mu$ Vision2 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
<b>Clear</b>	C	Clears the measurement record buffer.
<b>Display</b>	D	Displays the current time and input values.
<b>Time</b>	T <i>hh:mm:ss</i>	Sets the current time in 24-hour format.
<b>Interval</b>	I <i>mm:ss.ttt</i>	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).
<b>Start</b>	S	Starts the measurement recording. After receiving the start command, MEASURE samples all data inputs at the specified interval.
<b>Read</b>	R [ <i>count</i> ]	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.
<b>Quit</b>	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
000233DE 76E30004 OR
173: DP3 &= 0xF7FF;
000233E2 66E3FFF7 AND D
174: SOTIC = 0x80;
000233E6 E6B68000 MOV S
175: SORIC = 0x00;
000233EA E6B70000 MOV S
176: SOBG = 0x40;
000233EE E65A4000 MOV S
177: SOCON = 0x8011;
178: #endif
179:
180: /* setup the timer
000233F2 E6D81180 MOV S
181: TOREL = 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 00 00 00 00 00 00 00 00 00 00
00021022: 00 00 00 00 00 00 00 00 00 00 00 00
0002102E: 00 00 00 00 00 00 00 00 00 00 00 00
0002103A: 00 00 00 00 00 00 00 00 00 00 00 00
00021046: 00 00 00 00 00 00 00 00 00 00 00 00
00021052: 00 00 00 00 00 00 00 00 00 00 00 00
0002105E: 00 00 00 00 00 00 00 00 00 00 00 00
0002106A: 00 00 00 00 00 00 00 00 00 00 00 00
00021076: 00 00 00 00 00 00 00 00 00 00 00 00
00021082: 00 00 00 00 00 00 00 00 00 00 00 00
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

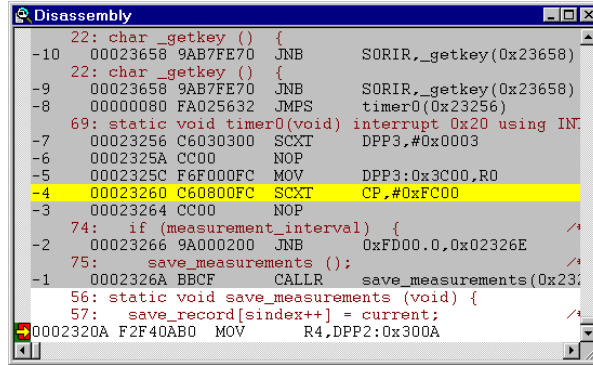
µVision2 internally tracks function nesting as the program executes. The **Call Stack** page of the **Watch Window** shows the current function nesting. A double click on a line displays the source code that called the selected function.

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

Locals | Watch #1 | Watch #2 | Call Stack

## 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 by reaching the breakpoint. The **Regs** page of the **Project Window** shows the CPU register contents for the selected instruction.

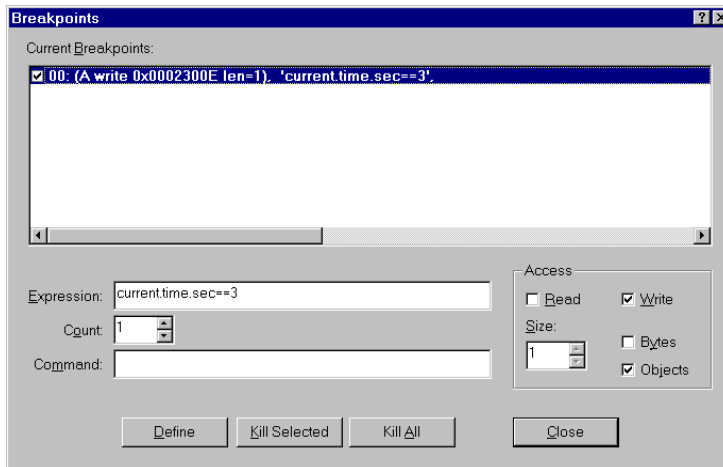


```

Disassembly
-10 00023658 9AB7FE70 JNB SORIR,_getKey(0x23658)
22: char _getKey () {
-9 00023658 9AB7FE70 JNB SORIR,_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 BBFC 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 69. 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**.



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,  $\mu$ Vision2 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:

- e 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.
- e 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**.
- e You can enter **WatchSet** in the **Output Window – Command** page.

Name	Value
\measure\index	0x0062
current	struct msec { ... }
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

$\mu$ Vision2 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:

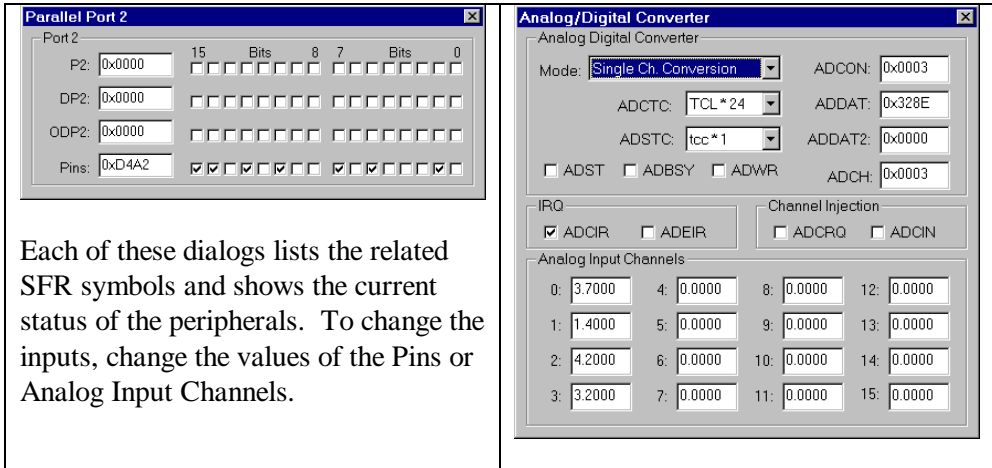
```

Serial #1
Command: d
Display current Measurements: (ESC to abort)
Time: 0:01:06.569 P2:D4A2 AN0:3.39V AN1:1.40V AN2:4.19V AN3:3.19V
  
```

You may now use the following procedures to supply input to the I/O pins:

### Using Peripheral Dialog Boxes

$\mu$ Vision2 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:Port2 and A/D Converter. The dialogs show the current status of the peripherals and you may directly change the input values.



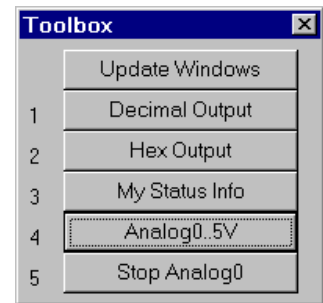
## Using VTREG Symbols

You may use the “CPU Pin Registers (VTREGs)” described on page 86 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:

```
PORT2=0xDA00      set digital input PORT2 to 0xDA00.
AIN1=3.3          set analog input AIN1 to 3.3 volts.
```

## Using User and Signal Functions

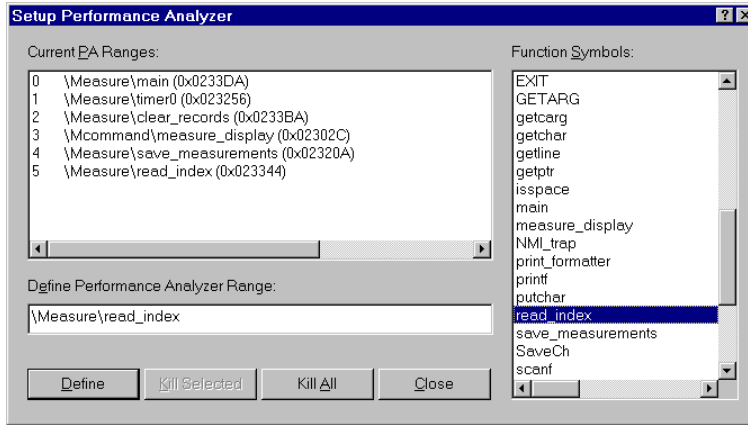
You may combine the use of VTREG symbols defined by the CPU driver and  $\mu$ Vision2 user and signal functions to create a sophisticated method of providing external input to your target programs. The “Analog Example” on page 108 shows a signal function that provides input to AIN0. The signal function 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.



7





## Using the Performance Analyzer

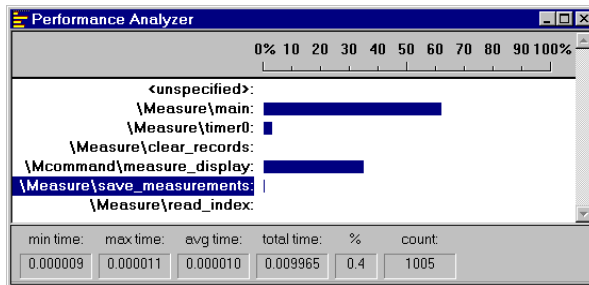
$\mu$ Vision2 lets you perform timing analysis of your applications using the integrated performance analyzer. To prepare for timing analysis, halt program execution and open the Setup **Performance Analyzer** dialog with the **Debug** menu.



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**



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 77 for more information.

## Chapter 8. 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 8xC166 and C167 family. Many of these are described in this chapter. You may use the code examples provided here to quickly get started working with the 166.

The on-chip peripherals of the C167 are quite sophisticated and offer numerous configuration options. The code examples in this chapter only present one or two methods of using each peripheral. Be aware that there are more configuration options than are presented in this text.

Topic	Page
"	128
Header Files"	
"DPP Registers"	129
"Interrupt"	131
"Peripheral Event Controller"	134
"Parallel Port I/O"	138
"General Purpose Timers"	140
"Serial Interface"	142
"Watchdog Timer"	145
"Pulse Width Modulation"	146
"A/D Converter"	149
"Power Reduction Modes"	150

---

### **NOTE**

*The code examples presented here were tested with the C167CR microcontroller. It is a simple procedure to convert them to work with your particular device. Make sure that the on-chip peripherals you attempt to use actually exist in your device.*

---

## Header Files and Startup Code

The on-chip peripherals of the 166/ST10 are accessed using the special function registers located in the system area of the on-chip memory. The Keil development tools provide include files or header files that define these registers. You must include the provided header files or you may create and include your own header files to access the on-chip peripherals.

Many of the example programs presented in this chapter begin with:

```
#include <reg167.h>
```

The **REG167.H** include file is located in the folder **C:\KEIL\C166\INC**. It defines the special function registers for the Siemens C167 device series. The following excerpt from that file shows the definitions for the parallel I/O ports.

```
/* I/O Ports */
sfr DP0L = 0xF100;
sfr DP0H = 0xF102;
sfr DP1L = 0xF104;
sfr DP1H = 0xF106;
sfr DP2 = 0xFFC2;
sfr DP3 = 0xFFC6;
sfr DP4 = 0xFFCA;
sfr DP6 = 0xFFCE;
sfr DP7 = 0xFFD2;
sfr DP8 = 0xFFD6;
```

It is easy to create your own custom header files for new chips or for your own projects.

The following table gives you an overview of the 166/ST10 derivatives currently available and the required startup and SFR definition files.

Derivatives	Instruction Set	Startup File	Include File
8xC166 / ST10x166	Standard 166	STARTUP.A66	REG166.H
C161	Enhanced 167	START167.A66	REG161.H
C163 / ST10x163	Enhanced 167	START167.A66	REG163.H
C164	Enhanced 167	START167.A66	REG164.H
C165 / ST10x165	Enhanced 167	START167.A66	REG165.H
C167 / ST10x167	Enhanced 167	START167.A66	REG167.H

Many devices include CAN peripherals. The CAN.H header file supports these peripherals.

## DPP Registers

In the segmented mode of the 166, the C166 compiler uses the four DPP registers (DPP0 - DPP3) to efficiently access memory with 16-bit near addresses. This is faster than full 32-bit memory accesses.

---

### NOTE

*Except for the **TINY** memory model, the C166 Compiler always uses the segmented mode of the 166/ST10.*

---

By default, DPP registers are allocated according to the following table.

DPP Register	By default, used in C programs
DPP0	To access <b>far</b> , <b>huge</b> , and <b>xhuge</b> objects code for the 8xC166 is generated. For CPU with extended instruction set, the compiler accesses <b>far</b> , <b>huge</b> and <b>xhuge</b> objects with EXTP and EXTS instructions instead of using the DPP0 register.
DPP1	The access the <b>NCONST</b> group (variables defined with <b>const near</b> ).
DPP2	The access the <b>NDATA</b> group (variables defined with <b>near</b> ).
DPP3	The access the <b>SDATA</b> or <b>SYSTEM</b> group (variables defined with <b>sdata</b> , <b>idata</b> or <b>bdata</b> ).

In  $\mu$ Vision2, you may specify under **Options – Target - Near Memory** the size for **RAM** (= **NDATA** group) and **ROM** (= **NCONST** group). If you specify more than 16KB for RAM or ROM,  $\mu$ Vision2 inserts the C166 **NOFIXDPP** and the L166 **DPPUSE** directive. **DPPUSE** reassigns the DPP registers for **NDATA** and **NCONST**. With **DPPUSE** the **NDATA** and **NCONST** groups can be expanded to a total of 64 Kbytes.

---

### NOTES

*The DPP3 register is also used when generating 166 code to speed-up memory and string copy and compare functions, if two far or huge objects are accessed.*

*When C166 is invoked with the MOD167 directive the DPP registers are never altered by C code or the run-time library.*

---

Typically, the 166 microcontroller family uses 16KB pages to access data. The DPP registers contain the base address for four different data pages. This has the advantage that the CPU instructions require only 16-bit address fields for memory accessing. Except the 8xC166 CPU, all 166/ST10 microcontrollers are using the extended instruction set which offer EXTS and EXTP instructions to overwrite the DPP addressing mode.

When using the C166 *near* addressing, the CPU gets a 14-bit offset and a 2-bit DPP selector. The top two bits of a 16-bit *near* address indicate the DPP which holds the base address for the 14-bit offset. For example, the assembler instructions below will use the DPP2 to form the physical address, since the top two bits of the instruction address 8042H are **10** indicating that DPP2 holds the base address for the 14-bit offset 0042H.

```
MOV    R1,0x8042    ; load R1 with memory location (DPP2*0x4000) + 0x0042
```

If DPP2 contains 2, the base will be  $2 * 0x4000 = 0x8000$ . Thus R1 will be loaded with the content of the memory address 0x8042. However, if DPP2 = 8, the instruction accesses the address:  $8 * 0x4000 + 0x0042H = 0x20042$ .

The same address calculations are performed for indirect addressing modes:

```
; R2 contains the value 0x4010H which indicates DPP1:
MOV    R1,[R2]      ; load R1 with memory location (DPP1*4000H) + 0010H
```

The EXTS and EXTP instructions overwrite the DPP addressing mode. In EXTS code sequences, the CPU uses linear 24-bit addresses; the upper 8-bits are specified by the EXTS instruction, the lower 16-bits are given by the executed instructions. An EXTP instruction sequence specifies a 10-bit page address; thus the CPU ignores the DPP contents and uses the base address given by the EXTP instruction instead. This is exemplified in the following code example:

```
EXTS   0x10,#1      ; use segment address 0x10 for the next instruction
MOV    R1,0x8042    ; load R1 with memory location 0x108042

; R2 contains 0x180, R3 contains 5:
EXTP   R3,#2        ; use page address in R3 for next two instructions
MOV    R4,[R2]      ; load R4 with location (4*0x4000+0x180) = 0x14180
MOV    R5,[R2+#2]   ; load R5 with location (4*0x4000+0x180+2) = 0x14182
```

The C166 compiler and the L166 Linker/Locator handles all the CPU addressing modes for you. Therefore the different CPU address calculations are *totally transparent* to the C programmer. Only during the debugging phase you need to know about DPP registers and EXTP / EXTS CPU instructions when you view pointer values or assembly code. You are faced with the 14-bit offset / 2-bit DPP selector format when *near* pointer values are displayed. A *far* pointer contains a 14-bit offset and a 10-bit page address. *Only during debugging*, you need to calculate the physical memory address for *near* and *far* pointer values as shown in the above examples. The *huge* and *xhuge* pointer values are representing directly physical memory addresses.



## Interrupts

The C166 compiler lets you write interrupt service routines in C. The compiler generates very efficient entry and exit code and accommodates register bank switching. Interrupt routines are declared as follows:

```
void function (void) interrupt vector [using rbank]
```

*function* is the name of the interrupt function.

*vector* is the interrupt vector definition.

*rbank* is the register bank name.

Typical interrupt routines are define as follows:

- The *vector* is only a trap number without any symbolic name.

```
void isr (void) interrupt 42 using RBANK1
```

- The *vector* is a symbolic name followed by the trap number it references.

```
void isr (void) interrupt S0TINT=42 using RBANK1
```

The *rbank* is a symbolic register bank name you define for a new register bank. The linker automatically reserves space for the register bank and the compiler automatically switches register bank contexts inside the interrupt routine. You may use the same register bank for interrupt routines which cannot interrupt each other. For example, you may define RBANK1 for interrupt priority level 1 (ILVL 1), RBANK2 for ILVL 2, and so on.

The following example code shows the interrupt code for the serial transmit interrupt routine.

```
1 void serial_TX_irq (void) interrupt S0TINT=42 using rbank1 {
2     1
3     1     if (tx_in != tx_out)           /* buffer not empty? */
4     1         S0TBUF = tx_buf [tx_out++]; /* transmit next character */
5     1     else
6     1         tx_restart = 1;           /* re-start transmit */
7     1 }
```

The following listing shows the code generated by the C166 compiler for the above interrupt routine. Note that the register bank context is swapped on entry to the interrupt routine and is restored on exit.

```

0000 C6030300      SCXT   DPP3,#03H      ; SOURCE LINE # 1
0004 CC00          NOP
0006 F6F00000 R   MOV    rbank1,R0
000A C60800C0 R   SCXT   CP,#rbank1   ; *** Switch to rbank1 ***
000E CC00          NOP
                                ; SOURCE LINE # 3
0010 F3F80002 R   MOVB   RL4,tx_out
0014 43F80202 R   CMPB   RL4,tx_in
0018 2D0B          JMPR    cc_Z,?C0001
                                ; SOURCE LINE # 4
001A F3FA0002 R   MOVB   RL5,tx_out
001E 258F0002 R   SUBB   tx_out,ONES
0022 C0A4          MOVBZ  R4,RL5
0024 F4840000 R   MOVB   RL4,[R4+#tx_buf]
0028 C084          MOVBZ  R4,RL4
002A F6F4B0FE     MOV    S0TBUF,R4
002E 0D01          JMPR    cc_UC,?C0002
0030              ?C0001:
                                ; SOURCE LINE # 6
0030 0F00          R   BSET   tx_restart
0032              ?C0002:
                                ; SOURCE LINE # 7
0032 FC08          POP    CP
                                ; *** Restore Register Bank ***
0034 FC03          POP    DPP3
0036 FB88          RETI

```

**NOTE**

If interrupt routines are small, it may be more efficient to exclude the **using** attribute and allow the compiler to push the registers used onto the stack. Therefore you should compare the assembler code generated by C166 for simple interrupt functions with and without **using** attribute.

## Interrupt Control Registers

The C167 provides interrupt services for nearly every on-chip peripheral. Interrupts are globally enabled and disabled using the **IEN** bit of the **PSW**. When **IEN** is set to 1, interrupts are enabled. When **IEN** is set to 0, interrupts are disabled.

Interrupts are individually controlled through the interrupt control register for each interrupt source. All interrupt control registers of the C167 have the same format as shown in the following figure.

Interrupt Control Register Layout

Interrupt Control Register Layout															
Reserved								xxIR	xxIE	ILVL				GLVL	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bits 15-8 are unused while bits 7-0 contain four fields that control different aspects of the interrupt. These fields are described in the following table.

Field	Description
<b>xxIR</b>	The interrupt request flag indicates whether or not an interrupt request is pending. A value of 1 indicates that an interrupt is pending while a value of 0 indicates that no interrupt is pending. This bit is written by hardware when the interrupt occurs and is cleared automatically by the hardware when the interrupt service routine exits. It may be set by software to trigger an interrupt service routine.
<b>xxIE</b>	The interrupt enable bit enables or disables an individual interrupt source. If set to 1, the interrupt is enabled. If set to 0, the interrupt is disabled.
<b>ILVL</b>	The interrupt priority level defines the priority level of the interrupt request. It may be a number from 0 to 15. Higher numbers represent higher priority levels. Note that an ILVL of 0 never gets serviced! However, priority level 0 may be used to awaken the CPU from IDLE mode. See "Idle Mode" on page 150 for more information.
<b>GLVL</b>	The group level defines the order in which simultaneous interrupts with the same priority level are services. The group level may be from 0 to 3. Higher numbers represent higher priority levels.

---

#### **NOTE**

*The **ILVL** and **GLVL** fields may not be the same for more than one interrupt source. Each enabled interrupt source must have its own unique combination of **ILVL** and **GLVL**.*

---

Since all interrupt control registers share the same format, you may define C macros to initialize the fields for the interrupt. The following code exemplifies the setup of the interrupt control registers.

```
#define IC_IE(x)      (((x) == 0) ? 0x0000 : 0x0040)
#define IC_ILVL(x)   (((x) << 2) & 0x003C)
#define IC_GLVL(x)   ((x) & 0x0003)

T3IC = IC_IE(1) | IC_ILVL(1) | IC_GLVL(0); /* Interrupt Enabled, Level 1 */
```

## Peripheral Event Controller

The 166 devices provide an 8-channel peripheral event controller or PEC that you can program to automatically move data (8-bit bytes or 16-bit words) when an interrupt occurs. The benefit of using the PEC is that the memory transfers are fast (only 1 CPU cycle) and the setup required is trivial.

The number of uses for the PEC is virtually unlimited. For example, you can program the PEC to...

- Read data from the A/D converter and store it in a buffer,
- Transfer data from a static buffer out to the serial port,
- Periodically read A/D input values and output them to the serial port.

Basically, the PEC works just like direct memory access (DMA) controllers. But, it is easier to program and it works much faster.

PEC transfers may occur only in page 0—the first 64K address space (0x000000-0x00FFFF). The PEC can read from and write to the special function registers of the CPU as well as to internal RAM.

Each PEC channel includes a PEC control register (PECC0-PECC7), a source pointer register (SRCP0-SRCP7), and a destination pointer register (DSTP0-DSTP7).

There are 4 steps you must follow to properly initialize and use the PEC.

1. Initialize the PEC Control Register.
2. Initialize the Source Pointer.
3. Initialize the Destination Pointer.
4. Initialize the Interrupt Control Register for the interrupt source.

Each of these steps is described in detail in the following sections.

### PEC Control Register

The PEC control register provides three fields that let you select whether bytes or words are moved (**BWT** field); whether the source pointer, destination pointer, or neither pointer are incremented after the move (**INC** field); and the number of

times to move the data (**COUNT** field). The layout of the control register is shown in the following figure.

PEC Control Register Layout															
Reserved					INC		BWT	COUNT							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bits 15-11 of each PEC control register are unused. Bits 10-0 contain three fields that control several aspects of the interrupt and PEC channel. These fields are described in the following table.

Field	Description
<b>INC</b>	The <b>INC</b> field selects the pointer that is incremented after the PEC transfer. Either the source pointer or the destination pointer may be incremented. Incrementing both pointers is not supported. Valid values are: 0 0: Neither pointer is modified. 0 1: Destination pointer ( <b>DSTP<sub>x</sub></b> ) is incremented by 1 (BWT=1) or by 2 (BWT=0). 1 0: Source pointer ( <b>SRCP<sub>x</sub></b> ) is incremented by 1 (BWT=1) or by 2 (BWT=0).
<b>BWT</b>	Selects whether bytes (when 1) or words (when 0) are transferred.
<b>COUNT</b>	Selects the number of transfers. When <b>COUNT</b> is 0xFF, continuous moves are made. When <b>COUNT</b> is a non-zero value, that number of moves are made. When <b>COUNT</b> is 0, the interrupt service routine is invoked instead of a PEC transfer.

## Source and Destination Pointers

The source and destination pointers are initialized directly in the C source code. The address is a 16-bit address that corresponds to the lower 64K of the C167. For example, the following line of code:

```
DSTP0 = (unsigned int) &S0TBUF; /* Destination is the serial output */
```

Assigns the value 0xFE00 to the destination pointer for PEC channel 0. This address is the address of the **S0TBUF** register (which is the serial 0 transmit buffer).

The following line of code:

```
SRCP0 = _sof_(string1); /* Source is STRING1 */
```

Assigns the address of **string1** to the source pointer for PEC channel 0.

### NOTE

Use the `_sof_` intrinsic library routine to obtain the page 0 offset of variables used with the PEC source and destination pointers.

## Channel Selection

A PEC channel is selected for a particular interrupt source through the interrupt control register. At most, 8 interrupts can use the PEC (since there are only 8 PEC channels). The interrupt control registers of the C167 use the format shown in the following figure when the PEC is enabled.

Interrupt Control Register Layout for PEC															
Reserved								xxIR	xxIE	1	1	1	PEC Channel		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bits 15-8 of each interrupt control register are unused. Bits 7-0 contain three fields that control several aspects of the interrupt and PEC channel. These fields are described in the following table.

Field	Description
<b>xxIR</b>	The interrupt request flag indicates whether or not an interrupt request is pending. A value of 1 indicates that an interrupt is pending while a value of 0 indicates that no interrupt is pending. This bit is written by hardware when the interrupt occurs and is cleared automatically by the hardware when the interrupt service routine exits. It may be set by software to trigger an interrupt service routine.
<b>xxIE</b>	The interrupt enable bit enables or disables an individual interrupt source. If set to 1, the interrupt is enabled. If set to 0, the interrupt is disabled.
<b>PEC Channel</b>	The PEC channel defines which PEC channel is used for the interrupt source.

Since all interrupt control registers share the same format for the PEC, you may define C macros to setup the fields. The following code exemplifies the setup of the interrupt control register for PEC channel transfers.

```
#define IC_IE(x)      (((x) == 0) ? 0x0000 : 0x0040)
#define IC_PEC(x)    (((x) & 0x0007) | 0x0038)
S0TIC = IC_IE(1) | IC_PEC(0);      /* Interrupt Enabled, PEC Channel 0 */
```

## PEC Example Program

The following example program shows how to program PEC channel 0 for data transfers to the serial port transmit buffer. It transmits the string “0123456789\n” out the serial port and then generates a transmit interrupt (**serial\_TX\_IRQ**). This interrupt function resets the PEC 0 transfer and the string is sent again.

```
#include <reg167.h>
#include <intrins.h>

#pragma PECDEF (0)      /* Reserve space for PEC channel 0 pointers */

const char string1 [] = "0123456789\n";

/* This function initializes PEC Channel 0 to move the data from */
/* STRING1 to the serial transmit buffer. */

void serial_PEC0_setup (void) {
    PECC0 = 0x0500          /* Move Bytes, Inc Src Ptr */
        | ((sizeof (string1) / sizeof (string1 [0])) - 1);

    SRCP0 = _sof_ (string1); /* Source is STRING1 */
    DSTP0 = (unsigned int) &SOTBUF; /* Destination is the serial output */
}

/* The Serial TX interrupt just resets PEC 0 and transfers another */
/* copy of STRING 1. */

void serial_TX_irq (void) interrupt SOTINT = 42 {
    serial_PEC0_setup ();
}

/* The setup routine for the serial port also initialized the PEC 0 */
/* transfer and sets a TX interrupt request */

void serial_setup (unsigned int baud) {
    /* Calculate reload value for baudrate generator */
    SOBGR = (20000000UL / (32UL * (unsigned long) baud)) - 1;
    DP3 |= 0x0400; /* Set TXD for output */
    DP3 &= ~0x0800; /* Set RXD for input */
    P3 |= 0x0400; /* Set TXD high */
    SOCON = 0x8011;
    SOTIC = 0x00F8; /* Serial TX IRQ = Level 14, Priority 0 (PEC 0) */
    serial_PEC0_setup ();
    IEN = 1; /* Enable interrupts */
}

void main (void) {
    serial_setup (19200);

    while (1) {
    }
}
```

## Parallel Port I/O

The C167 provides a number of parallel I/O ports you may use for your target application. Many may be used for general purpose I/O. However, some ports have alternate uses that may prevent their generic use by your application.

Port	Direction	Width	Alternate Use
<b>P0L</b>	I/O	8 bits	Mux'd. 16-bit bus: A0-A7 & D0-D7 Mux'd. 8-bit bus: A0-A7 & D0-D7 Non-Mux'd. 16-bit bus: D0-D7 Non-Mux'd. 8-bit bus: D0-D7
<b>P0H</b>	I/O	8 bits	Mux'd. 16-bit bus: A8-A15 & D8-D15 Mux'd. 8-bit bus: A8-A15 Non-Mux'd. 16-bit bus: D8-D15 Non-Mux'd. 8-bit bus: Available for user I/O
<b>P1L</b>	I/O	8 bits	Mux'd. 16-bit bus: Available for user I/O Mux'd. 8-bit bus: Available for user I/O Non-Mux'd. 16-bit bus: A0-A7 Non-Mux'd. 8-bit bus: A0-A7
<b>P1H</b>	I/O	8 bits	Mux'd. 16-bit bus: Available for user I/O Mux'd. 8-bit bus: Available for user I/O Non-Mux'd. 16-bit bus: A8-A15 Non-Mux'd. 8-bit bus: A8-A15 P1H.4-P1H.7: Compare/Capture I/O
<b>P2</b>	I/O	16 bits	P2.0-P2.15: Compare/Capture I/O
<b>P3</b>	I/O	15 bits	P3.1: Timer 6 Output P3.2: CAPIN P3.3: Timer 3 Output P3.4: Timer 3 Ext. Up/Down P3.5: Timer 4 Input P3.6: Timer 3 Input P3.7: Timer 2 Input P3.10: Serial Chan. 0 Transmit P3.11: Serial Chan. 0 Receive P3.12: /BHE or /WRH P3.13: SSC Master Clock Output P3.14: Not Implemented P3.15: System Clock Output
<b>P4</b>	I/O	8 bits	P4.0-P4.7: A16-A23 P4.4: CAN2_RxD P4.5: CAN1_RxD P4.6: CAN1_TxD P4.7: CAN2_TxD
<b>P5</b>	I	16 bits	P5.0-P5.15: A/D Analog Inputs P5.10: Timer 6 Ext. Up/Down P5.11: Timer 5 Ext. Up/Down P5.12: Timer 6 Input P5.13: Timer 5 Input P5.14: Timer 4 Ext. Up/Down P5.15: Timer 2 Ext. Up/Down



Port	Direction	Width	Alternate Use
P6	I/O	8 bits	P6.0-P6.4: CS0-CS4 Output P6.5: External Master Hold Req. Input P6.6: Hold Acknowledge Output P6.7: Bus Request Output
P7	I/O	8 bits	P7.0-P7.3: PWM Outputs 0-3 P7.4-P7.7: Compare/Capture I/O
P8	I/O	8 bits	P8.0-P8.7: Compare/Capture I/O

Most ports have data direction registers you must properly initialize to specify whether each port pin is an input or an output. The pins of Port 2, Port 3, Port 6, Port 7, and Port 8 may be programmed for push-pull operation or open-drain operation via control registers. The following table specifies the names for these registers.

Port Register	Direction Control Register	Push-Pull/Open-Drain Control Register
P0L	DP0L	
P0H	DP0H	
P1L	DP1L	
P1H	DP1H	
P2	DP2	ODP2
P3	DP3	ODP3
P4	DP4	
P6	DP6	ODP6
P7	DP7	ODP7
P8	DP8	ODP8

*Write a 1 to a direction control register bit to configure the corresponding port bit or pin as an output. Write a 0 to configure the port bit or pin as an input.*

*Write a 1 to a push-pull/open-drain control register bit to configure the corresponding port bit or pin as an open-drain output. Write a 0 to configure the port bit or pin as a push-pull output.*

The following example program shows how to output values to Port 7 in open-drain configuration.

```
#include <reg167.h>

void main (void) {
    unsigned int i;

    DP7 = 0xFF;          /* Setup P7.7-P7.0 for output */
    ODP7 = 0xFF;        /* Setup P7.7-P7.0 for open-drain */

    while (1) {
        for (i = 0x01; i <= 0x80; i <<= 1) {
            P7 = i;      /* Write new value to P7 */
        }
    }
}
```

## General Purpose Timers

The C167 has two groups (**GPT1** and **GPT2**) of general purpose timers/counters. GPT1 contains 3 timers/counters (T2, T3, and T4) and GPT2 contains 2 timers/counters (T5 and T6). Each timer in each group may operate independently in a number of different modes including timer mode, counter mode, gated timer mode, reload mode, and capture mode.

The following table lists the special function registers used to program the general purpose timers.

Register	Description
<b>CAPREL</b>	GPT2 capture/reload register.
<b>CRIC</b>	GPT2 CAPREL interrupt control register.
<b>T2</b>	GPT1 timer 2 register.
<b>T3</b>	GPT1 timer 3 register.
<b>T4</b>	GPT1 timer 4 register.
<b>T5</b>	GPT2 timer 5 register.
<b>T6</b>	GPT2 timer 6 register.
<b>T2IC</b>	GPT1 timer 2 interrupt control register.
<b>T3IC</b>	GPT1 timer 3 interrupt control register.
<b>T4IC</b>	GPT1 timer 4 interrupt control register.
<b>T5IC</b>	GPT2 timer 5 interrupt control register.
<b>T6IC</b>	GPT2 timer 6 interrupt control register.
<b>T2CON</b>	GPT1 timer 2 control register.
<b>T3CON</b>	GPT1 timer 3 control register.
<b>T4CON</b>	GPT1 timer 4 control register.
<b>T5CON</b>	GPT2 timer 5 control register.
<b>T6CON</b>	GPT2 timer 6 control register..

The following example program shows how to use Timer 3 to generate a 1000Hz timer tick interrupt for timing purposes. The timer tick interrupt increments the **timer\_tick** variable once for each interrupt. The **timer3\_setup** function initializes the timer and the **timer3\_delay** function delays for the specified number of timer ticks.

This example program toggles a bit on Port 7 every 0.100 seconds.

```

#include <reg167.h>
#include <intrins.h>

static unsigned long volatile timer_tick = 0UL;

static void timer3_irq (void) interrupt T3INT = 35 {
    timer_tick++;
}

void timer3_setup (unsigned int ticks_per_sec) {
    unsigned int reload;
    unsigned long frequency = 2500000;
    unsigned int prescale;

    for (prescale = 0; prescale < 8; prescale++) {
        if ((frequency / ticks_per_sec) <= 65535) break;
        frequency /= 2;
    }
    reload = frequency / ticks_per_sec;

    T3CON = 0x0080;    /* 2.5MHz, Timer Mode, Count Down */
    T3CON |= prescale;

    T2CON = 0x0027;    /* Setup T2 for reload operation on any T3OTL */
    T2 = reload;       /* Reload for T3 */
    T3 = reload;       /* Start T3 with proper reload */
    T3IC = 0x0044;    /* Timer 3 interrupt enabled, Level 1 */
    T3R = 1;          /* Start Timer */
    IEN = 1;          /* Enable Interrupts /
}

void timer3_delay (unsigned long ticks) {
    unsigned long start_tick;
    unsigned long timer_tick_copy;

    _atomic_ (0);      /* start un-interruptable code */
    start_tick = timer_tick_copy;
    _endatomic_ ();    /* end un-interruptable code */

    while (1) {
        _atomic_ (0);    /* start un-interruptable code */
        timer_tick_copy = timer_tick;
        _endatomic_ ();    /* end un-interruptable code */
        if ((timer_tick_copy - start_tick) > ticks) break;
    }
}

void main (void) {
    DP7 = 0x01;        /* Setup P7.0 for output */
    ODP7 = 0x01;      /* Setup P7.0 for open-drain */
    timer3_setup (1000); /* Setup timer for 1000Hz operation */
    while (1) {
        timer3_delay (100); /* Wait 0.10 seconds */
        P7 |= 0x01;        /* Turn on P7.0 */
        timer3_delay (100); /* Wait 0.10 seconds */
        P7 &= ~0x01;      /* Turn off P7.0 */
    }
}

```

## Serial Interface

The C167 includes a standard RS-232 compatible serial port (ASC0) you may use with your application programs. The C167 uses port pins P3.10 and P3.11 for transmit and receive respectively. You must properly configure these port pins for input and output to use the serial port. Additionally, there is a baudrate reload register and a control register that you must properly configure before serial port 0 will function.

The C167 provides full interrupt control for the serial port transmit, receive, and error conditions. There are separate transmit and receive buffers you write outgoing data to and read incoming data from. You may poll the interrupt control registers to determine when a character has been receive or when the next character may be sent. You may also create interrupt routines to handle these operations.

The following table lists the special function registers used to program serial port 0.

Register	Description
<b>S0BG</b>	Serial port 0 baud rate generator reload register.
<b>S0CON</b>	Serial port 0 control register.
<b>S0EIC</b>	Serial port 0 error interrupt control register.
<b>S0RBUF</b>	Serial port 0 receive buffer.
<b>S0TBIC</b>	Serial port 0 transmit buffer interrupt control register.
<b>S0TBUF</b>	Serial port 0 transmit buffer.
<b>SORIC</b>	Serial port 0 receive interrupt control register.
<b>SOTIC</b>	Serial port 0 transmit interrupt control register.

The following example program shows how to perform interrupt-driven serial I/O using the C167's asynchronous serial channel. Interrupt routines in this example handle transmit interrupts (**serial\_TX\_irq**) and receive interrupts (**serial\_RX\_irq**) using 256-byte circular buffers. Routines are provided to transmit (**serial\_TX**) and receive (**serial\_RX**) characters and to initialize the serial channel (**serial\_setup**). Additionally, the library routines for **putchar** and **\_getkey** have been replaced with ones that use the interrupt-driven serial I/O routines. This also lets the **printf** and **scanf** library functions work with the interrupt-driven I/O routines in this example.

```

#include <reg167.h>
#include <intrins.h>

static unsigned char volatile rx_buf [sizeof (unsigned char) * 256];
static unsigned char          tx_buf [sizeof (unsigned char) * 256];

/* Note: variables that are modified in interrupts are volatile! */
static unsigned char volatile rx_in  = 0; /* RX buffer next in index */
static unsigned char          rx_out = 0; /* RX buffer next out index */

static unsigned char          tx_in  = 0; /* TX buffer next in index */
static unsigned char volatile tx_out = 0; /* TX buffer next out index */
static bit volatile tx_restart = 0;      /* NZ for transmit restart */

void serial_TX_irq (void) interrupt S0TINT = 42 {
    if (tx_in != tx_out) /* buffer not empty? */
        S0TBUF = tx_buf [tx_out++]; /* transmit the next character */
    else tx_restart = 1; /* transmit must be re-started */
}

void serial_RX_irq (void) interrupt S0RINT = 43 {
    rx_buf [rx_in++] = S0RBUF; /* put received character in buffer */
}

char putchar (char c) {
    /* substitute function for putchar */
    /* wait while buffer is full */
    while (((unsigned char)(tx_in + 1)) == tx_out); /* buffer full? */
    _atomic_ (0); /* start un-interruptable code */
    tx_buf [tx_in++] = c; /* put the character in the buffer */
    _endatomic_ (); /* end un-interruptable code */
    if (tx_restart) { /* if transmits must be restarted... */
        tx_restart = 0; /* clear re-start flag */
        S0TIR = 1; /* enable transmit request */
    }
    S0TIE = 1; /* enable interrupt */
    return (c);
}

char _getkeyserial_RX (void) { /* substitute function for _getkey */
    while (rx_in == rx_out);
    return (rx_buf [rx_out++]);
}

void serial_setup (unsigned int baud) {
    /* Calculate reload value for baudrate generator */
    S0BG = (20000000UL / (32UL * (unsigned long) baud)) - 1;
    DP3 |= 0x0400; /* Set TXD for output */
    DP3 &= ~0x0800; /* Set RXD for input */
    P3 |= 0x0400; /* Set TXD high */
    S0CON = 0x8011;
    S0RIC = 0x0044; /* Enable serial receive interrupt lvl 1 */
    S0TIC = 0x0008; /* Disable serial transmit interrupt lvl 2*/
    tx_restart = 1; /* Set restart flag */
}

void main (void) {
    serial_setup (19200);
}

```

```
IEN = 1;          /* Enable interrupts */

printf ("Serial I/O Initialized\n");

while (1) {
    char c;

    c = getchar ();
    printf ("\nYou typed the character %c.\n", c);
}
}
```

## Watchdog Timer

The C167 includes a 16-bit watchdog timer for recovery from hardware or software failures. The watchdog timer counts up until it overflows or until it is reset by the **SRVWDT** instruction. If the watchdog timer overflows, it resets the CPU and starts executing your program from the beginning exactly as if a hardware reset occurred.

Your application must periodically reset the watchdog timer using the `_srvwdt_` intrinsic C library function. If your program does not reset the watchdog timer frequently enough or if your program crashes, the watchdog timer overflows and resets the CPU.

The watchdog timer configuration register **WDTCON** lets you specify the reload value for upper byte of the timer as well as the clock divisor (2 or 128).

The following example code shows how to initialize the watchdog timer and how to reset it.

```
#include <reg167.h>
#include <intrins.h>

void main (void) {
    WDTCON = 0x8001;          /* Setup the watchdog timer      */
    /*      80..-----      Set the reload value to 0x80 */
    /*      ...1-----      Divide by 2                */

    while (1) {
        /* Applicaiton Code */

        _srvwdt_ ();        /* Reset the watchdog timer */
    }
}
```

### NOTE

*The watchdog timer is enabled after a software reset, hardware reset, or watchdog reset. Your application must disable the watchdog timer if it is not used. By default, the C startup code disables the watchdog for you.*

However, if your application uses the watchdog timer, make sure the startup code (found in **START167.A66**) properly enables it. Look for the following section in the startup code and set the **WATCHDOG** variable to 1.

```
; WATCHDOG: Disable Hardware Watchdog
; --- Set WATCHDOG = 1 to enable the Hardware watchdog
$SET (WATCHDOG = 1)
;
```

## Pulse Width Modulation

Pulse width modulation (PWM) is a method of varying the width of a pulse to control the duty cycle of a signal. Dimmer switches use this technology to control the amount of the AC sine wave that reaches the light bulb.

The C167 provides 4 independent PWM channels. The PWM signals are output on Port 7 pins 0 to 3. Each channel has a 16-bit up/down counter, a 16-bit period register, and a 16-bit pulse width register. There are 2 common control registers and a common interrupt control register.

The PWM output signals are XORed with the outputs of their respective Port 7 latches. After reset, the Port 7 latches are cleared to 0 and the PWM signals go directly to the port pins. You may set a port latch to invert the PWM signal on that port pin. For example, setting P7.0 inverts the PWM channel 0 output signal.

Register	Description
<b>PPx</b>	PWM period register for channel x.
<b>PWx</b>	PWM pulse width register for channel x.
<b>PTx</b>	PWM counter register for channel x.
<b>PWMCON0</b>	PWM control register 0.
<b>PWMCON1</b>	PWM control register 1.
<b>PWMIC</b>	PWM interrupt control register.

The clock for each of the PWM counters is the CPU clock with no divisor or with a divisor of 64. In the standard PWM generation mode, the PWM counter (**PTx**) counts up from 0 until it reaches the defined period and then resets to 0.

If the clock frequency is 20.000 MHz and the PWM counter is driven with no divisor and the period register (**PPx**) contains 19999, the counter counts from 0 to 19999 (20,000 counts). 20,000,000 divided by 20,000 is 1,000 (1.000 kHz).

The pulse width register (**PWx**) contains the count at which the PWM signal is “**on**”. When the PWM counter contains a value that is less than the pulse width register, the output is “**off**” or 0. When the PWM counter contains a value that is greater than the pulse width register, the output is “**on**” or 1.

If the period is setup for 19999 (20000 counts), the following pulse widths generate the specified duty cycles.



Pulse Width	Duty Cycle <sup>†</sup>
0	100.0%
2500	87.5%
5000	75.0%
10000	50.0%
15000	25.0%
17500	12.5%
20000	0.0%

<sup>†</sup> For a period of 20,000 counts.

The following example code shows functions to initialize the PWM (**PWM\_setup**) and how to set the pulse width (**PWM\_pulse\_width**).

```
#include <reg167.h>

void PWM_setup (unsigned char channel, unsigned int period) {
    if (channel > 3) return;

    DP7 |= (1 << channel); /* Setup P7.channel for output */
    ODP7 |= (1 << channel); /* Setup P7.channel for open-drain */
    P7 &= ~(1 << channel); /* P1.channel = 0 */

    switch (channel) {
        case 0:
            PP0 = period; /* Set PWM period */
            PW0 = period + 1; /* Set 0% duty cycle */
            PB01 = 0; /* Channel 0 & 1 are independent */
            break;

        case 1:
            PP1 = period; /* Set PWM period */
            PW1 = period + 1; /* Set 0% duty cycle */
            PB01 = 0; /* Channel 0 & 1 are independent */
            break;

        case 2:
            PP2 = period; /* Set PWM period */
            PW2 = period + 1; /* Set 0% duty cycle */
            PS2 = 0; /* Standard mode (non-single shot) */
            break;

        case 3:
            PP3 = period; /* Set PWM period */
            PW3 = period + 1; /* Set 0% duty cycle */
            PS3 = 0; /* Standard mode (non-single shot) */
            break;
    }

    PWMCON0 |= (0x0001 << channel); /* Start the PTx counter */
    PWMCON0 &= ~(0x0010 << channel); /* PTx clocked with CLKCPU */
    PWMCON0 &= ~(0x0100 << channel); /* Disable interrupts */
    PWMCON0 &= ~(0x1000 << channel); /* Clear interrupt request */

    PWMCON1 |= (0x0001 << channel); /* Enable channel output */
    PWMCON1 &= ~(0x0010 << channel); /* Setup edge aligned mode */
}
```

```
void PWM_pulse_width (unsigned char channel, unsigned int width) {
    switch (channel) {
        case 0: PW0 = width; break;
        case 1: PW1 = width; break;
        case 2: PW2 = width; break;
        case 3: PW3 = width; break;
    }
}

void main (void) {
    PWM_setup (0, 20000 - 1); /* 20MHz/20000 = 1kHz (1ms pulse width) */

    while (1) {
        PWM_pulse_width (0, 0); /* 100% duty cycle (ON from 0 to 19999) */
        PWM_pulse_width (0, 20000); /* 0% duty cycle */
        PWM_pulse_width (0, 10000); /* 50% duty cycle */
        PWM_pulse_width (0, 15000); /* 25% duty cycle */
    }
}
```

## A/D Converter

The C167 A/D converter provides 16 channels of 10-bit analog to digital conversion. Voltages presented to the input pins on Port 5 are converted to digital values and may be read from the A/D result register. Analog inputs may range from the voltages present on the  $V_{AREF}$  and  $V_{AGND}$  pins.

The on-chip A/D converter may be configured for a number of conversion modes including single or continuous conversions on one or more analog input channels. In addition, the C167 A/D converter can generate interrupts for end-of-conversion and overwrite conditions and can even be used to trigger a PEC data transfer.

Register	Description
<b>ADCON</b>	A/D converter control register.
<b>ADDAT</b>	A/D converter result register.
<b>ADDAT2</b>	A/D converter channel injection result register.
<b>ADCIC</b>	A/D converter interrupt control register (for end-of-conversion).
<b>ADEIC</b>	A/D converter interrupt control register (for overrun errors and channel injection).

The following example code shows how to initialize the A/D converter for single channel single conversion mode and read the data for that channel.

```
#include <reg167.h>

unsigned int ADC_read (unsigned char channel) {
    ADCON = 0xB000;
    /*      B...----- Conversion Clock = TCC = 96 *TCL */
    /*      B...----- Sample Clock = 8 * TCC */

    ADCON |= channel & 0x000F; /* Select channel to convert */
    ADST = 1; /* Begin conversion */

    while (ADBSY == 1); /* Wait while the ADC is converting */
    return (ADDAT & 0x03FF); /* Return the result */
}

void read_adc_channels (void) {
    unsigned char i;

    while (1) {
        for (i = 0; i < 16; i++) { /* Loop thru ADC channels */
            printf ("ADC Channel %u = %u\n", /* Print channel */
                (unsigned) i, /* Print ADC input */
                (unsigned) ADC_read (i));
        }
    }
}
```

## Power Reduction Modes

The C167 offers two different power saving modes you may invoke using a single instruction: **Idle Mode** and **Power Down Mode**.

### Idle Mode

**Idle Mode** halts the CPU but lets peripherals continue operating. When any reset or interrupt condition occurs, **Idle Mode** is canceled. Power consumption can be greatly decreased in idle mode. All peripherals including the watchdog timer continue to operate normally.

To enter **Idle Mode**, your program must execute the **IDLE** instruction. You may do this directly in C using the `_idle_` intrinsic library function.

```
#include <intrins.h>

void main (void) {
    while (1) {
        task_a ();
        task_b ();
        task_c ();

        _idle_ ();           /* Enter IDLE Mode */
    }
}
```

Any interrupt condition, regardless of **IEN**, terminates **Idle Mode**. This applies only to those interrupts whose individual interrupt enable flags were set before **Idle Mode** was entered.

After the **RETI** instruction of the interrupt that terminates **Idle Mode**, the CPU begins executing the code following the **IDLE** instruction.

### Power Down Mode

**Power Down Mode** halts both the CPU and peripherals. It is canceled only by a hardware reset.

To enter **Power Down Mode**, your program must pull the **NMI** pin low using external hardware and execute the **PWRDN** instruction using the `_pwrdn_` intrinsic library function.

```
#include <intrins.h>

void main (void) {
    task_a ();
    task_b ();
```

```
task_c ():  
    _pwrdn_ ();          /* Enter POWER DOWN Mode - Wait for reset */  
}
```

If the **NMI** pin is not held low, the **PWRDN** instruction is ignored and the 166 does not go into **Power Down Mode**.

Once **Power Down Mode** is entered, only a hardware reset will restart the CPU and peripherals.



## Chapter 10. CPU and C Startup Code

Your target program must initialize the CPU to match the configuration of your hardware design. The `STARTUP.A66` file contains the startup code for an 8xC166 target program. The `START167.A66` file contains the startup code for all other 166 derivatives. These source files are located in the `\C166\LIB` directory. You should copy either of these files to your project directory and make changes to match your hardware configuration.

The startup code executes immediately upon reset of the target system. It optionally performs the following operations:

1. Initialize the `SYSCON` and `BUSCON` SFRs,
2. Initialize the `ADDRSELx` and `BUSCONx` SFRs,
3. Reserve and initialize the hardware stack and stack overflow and underflow SFRs,
4. Set `DPP0 - DPP3` and `CP` for memory and registerbank accesses,
5. Reserve and initialize the user stack area and user stack pointer (`R0`),
6. Clear data memory,
7. Initialize variables that are explicitly initialized in the C source code,
8. Transfer control to the `main` C function.

### Selecting the Memory Model

When you assemble the startup code outside of `µVision2`, you must tell the assembler which memory model you use. You do this with the `SET` command and the memory model: `TINY`, `SMALL`, `COMPACT`, `HCOMPACT`, `MEDIUM`, `LARGE`, or `HLARGE`. For example, if you use the `SMALL` memory model, use the following command line to assemble the startup code:

```
A166 START167.A66 SET (SMALL)
```

### Configuring the Startup Code

The `STARTUP.A66` and `START167.A66` files contain definitions at the beginning which are used for the chip hardware configuration and for the C run-time system. An overview of the groups of configuration statements is provided below.

## Hardware Configuration

Name	Description
<b>SYSCON</b>	CPU system control register. This configures the chip parameters like basic bus characteristics, power-down modes, chip select configuration, system clock parameters, and so on.
<b>SYSCON2</b> <b>SYSCON3</b>	Power-down control registers, CPU clock control registers, and on-chip peripheral enable registers (available only on some devices).
<b>BUSCON<math>n</math></b> <b>ADDRESS<math>n</math></b> <b>RANGE<math>n</math></b>	Initialization registers for BUSCON0-BUSCON4. These registers define the starting address, range, and bus characteristics for different memory devices (like FLASH, SRAM, EPROM, and I/O) accessed via the chip select outputs.

## C Compiler Run-Time Configuration

Name	Description
<b>CLR_MEMORY</b>	Memory Zero Initialization of RAM areas. Default: enable the memory zero initialization of RAM area. To disable the memory zero initialization enter “\$SET (CLR_MEMORY = 1)”; this reduces the code size of the startup code.
<b>DPPUSE</b>	Allow re-assignment of DPP registers. Default: 1 to support the L166 DPPUSE directive. To disable the DPP re-assignment enter “\$SET (DPPUSE = 0)”; this reduces the code size of the startup code.
<b>INIT_VARS</b>	Variable Initialization of explicit initialized variables (The variables are to be defined static or declared at file level). Default: initialize variables. To disable the variable initialization enter “\$SET (INIT_VARS = 0)”; this reduces the code size of the startup code.
<b>SSTSZ</b>	Set the actual stack space for the system stack, if you have selected 7 for the STK_SIZE.
<b>STK_SIZE</b>	STK_SIZE: Maximum System Stack Size selection initialization value. Default value is 0 for 256 words stack size. This is also the reset value. Set STK_SIZE to the following values for other stack sizes: 0 for 256 words system stack. 1 for 128 words system stack. 2 for 64 words system stack. 3 for 32 words system stack. 4 for 512 words system stack (not for 166) 7 for user defined size of the system stack
<b>USTSZ</b>	Set the actual stack space for the user stack. The user stack is used for automatic variables. The USTSZ variable allows you to set the size for the user stack.
<b>WATCHDOG</b>	Hardware Watchdog control. Default: disable the hardware watchdog. To enable the watchdog enter “\$SET (WATCHDOG = 1)”.

### NOTE

*Siemens offers a configuration and programming utility called DAVE. This free utility helps you configure the CPU and create C source code to use the on-chip peripherals of the various 166 derivatives.*



## Chapter 11. Using Monitor-166

The Keil Monitor-166 allows you to connect your 166/ST10 hardware to the uVision2 Debugger. You can use the powerful debugging interface to test application programs in your target hardware.

The Monitor requires that the program you are debugging is located in RAM space. To set breakpoints in your code, the Monitor inserts TRAP instructions at all breakpoint locations. This operation is completely transparent, but may have side effects when calculating program checksums.

The Monitor program requires two interrupt vectors: the NMI interrupt is used for breakpoints. You may break program execution with a switch connect to the NMI pin of the CPU. The serial interface requires an additional interrupt, to stop program execution with the uVision2 HALT toolbar command.

The Keil Monitor-166 may be configured in three different operating modes:

### Bootstrap Mode

In bootstrap mode, the Monitor program will be downloaded into RAM of the target system. This is typically the best mode to start working with the Monitor, since it auto-adjusts baudrates and does not require burning any EPROM's. The Monitor communicates with the 166/ST10 built-in UART ASC0.

### UART Mode

In this configuration the Monitor program will be directly programmed to (Flash) EPROM's. It also communicates via the UART ASC0 (or ASC1 for 8xC166 CPU) and requires exact configuration of the baudrate. Compared to the **Bootstrap** mode, you save the time to download the Monitor program at system startup.

### Simulated Serial Mode

This configuration uses a *simulated serial interface* and does not require the 166/ST10 on-chip UART. You may use two unused I/O pins of the 166/ST10 to establish the communication between uVision2 and your target hardware. This mode does not use any of the on-chip peripherals, but has the restrictions that you cannot use the uVision2 HALT toolbar command, since the serial interrupt is not available.

## Bootstrap Loader

Many derivatives of the 166 include a bootstrap loader (BSL) that uses the serial port to download code to the on-chip RAM. The asynchronous serial port (ASC0) is used to transfer the program code.

For example, the C167 device enters BSL mode when port pin P0L.4 is low at the end of the hardware reset. After entering BSL mode, the C167 waits to receive a zero byte on the RXD0 line. The zero byte must contain 1 start bit, eight zero bits, and a stop bit. Once the zero byte is received, the BSL initialized the ASC0 interface with the appropriate baudrate and transmits an identification byte on TXD0. This procedure is similar for other 166 derivatives.

The bootstrap loader is used by the Keil Monitor to download itself and your program for debugging. Your target board does not require a monitor ROM. Only RAM devices are required for testing your programs.

## Hardware and Software Requirements

The following requirements must be met for Monitor-166 to operate correctly:

- Siemens 161/163/164/165/166/167 CPU or ST10 variant
- Serial interface for communication with the PC.
- Software trap used for breakpoints (usually NMI trap).
- Additional 10 words stack space in the user program to be tested.
- 256 bytes off-chip data memory (RAM).
- 5 Kbytes of-chip code memory loaded with Monitor-166 software (ROM or RAM in Bootstrap Mode).

All other hardware components can be used by the application.

# 8

## Serial Transmission Line

Monitor-166 requires only the signals **TRANSMIT DATA**, **RECEIVE DATA** and **SIGNAL GROUND** from the RS232 or V.24 line. However, in most cases, some additional connections are necessary in the serial connectors, to enable transmit and receive data.

PIN connections of various computer systems

## 25 Pin Connector

Signal Name	Pin	Description
RxD	3	receive data
TxD	2	transmit data
Gnd	7	signal ground

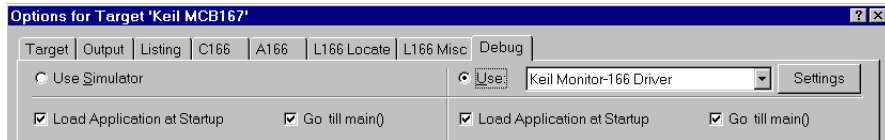
## 9 Pin Connector

Signal Name	Pin	Description
RxD	2	receive data
TxD	3	transmit data
Gnd	5	signal ground

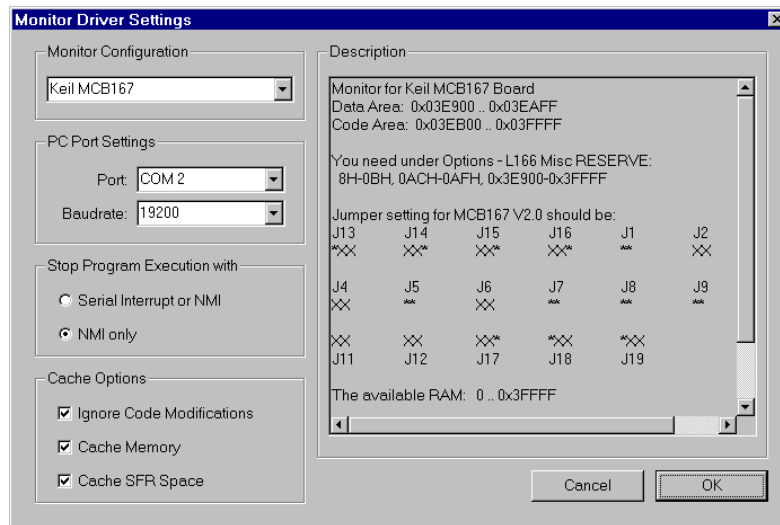
In addition to the above pins, connect pin 7 to pin 8 and pin 1 to pin 4 and pin 6.

## µVision2 Monitor Driver

µVision2 interfaces to target systems when you select **Use: Keil Monitor-166 Driver** in the dialog **Options – Debug**.



Click on **Settings** to open the dialog **Monitor Driver Settings** that allows you to configure various parameters such as COM port and baudrate. Refer to “Set Debug Options” on page 75 for more information about the Debug dialog.



The following table describes the **Monitor Driver Settings** page:

Dialog Item	Description
Monitor Configuration	List available Monitor configurations. You may add user configurations as described under "Monitor-166 Configuration" on page 160. Select the Monitor configuration for your target hardware.
Description	Provides you with a quick description of you target hardware. It contains also required configuration settings. You may use cut and paste to copy the settings into other Options dialog pages.
PC Port Settings	Select the PC COM port and the baudrate you want to use. If you have problems with your target hardware, try the Baudrate 9600.
Stop Program Execution with	When <b>Serial interrupt or NMI</b> is enabled, you can terminate a running application program with the <b>Stop</b> toolbar button or the <b>ESC</b> key in the Command page. To support this, the serial interface is not longer available for the user program. In addition, it is not allowed to reset the global Interrupt Enable Flag IE (bit in PSW) in your application.  In any case a high to low transition at the NMI# pin terminates a running application.
Cache Options	To speed up the screen updates, the Monitor driver implements several data caches.
Ignore Code Modifications	When enabled $\mu$ Vision2 duplicates the program code on the PC and never reloads the code from your target system. You should disable this option to debug self-modifying code.
Cache Memory	enables the memory cache for data regions. When you single step trough code, $\mu$ Vision2 will reload the data regions, even when memory cache is enabled. You should disable this option to view changes on IO ports or peripherals when debugging is stopped.
Cache SFR space	enables the memory cache for the SFR space of the CPU. When you single step trough code, $\mu$ Vision2 will reload the SFR regions, even when SFR cache is enabled. You should disable this option to view changes on IO ports or peripherals when debugging is stopped.

## Restrictions of $\mu$ Vision2 when using Monitor-166

The **memory mapping** of a CPU board with Monitor-166 is selected with hardware components and the Monitor configuration file. It is not possible to use **Debug – Memory Map** to change the memory mapping of the target system.

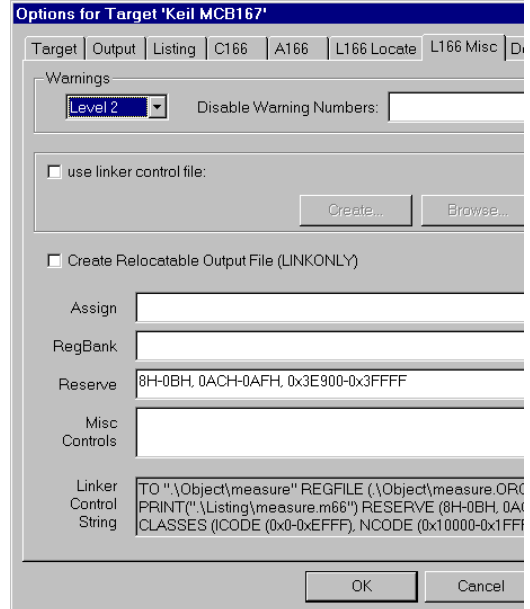
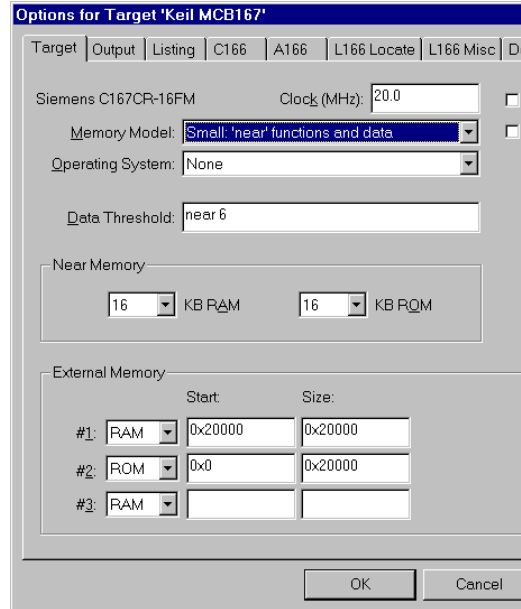
The **Performance Analyzer**, **Call Stack** and **Code Coverage** features are not available with Monitor-166.

**Breakpoint Options** are handled directly by Monitor-166. However, when **access** or **conditional** breakpoints are set, the application is executed in single steps and not in real time. Single step execution is at least 1000 times slower.

## Target Options when Using Monitor-166

When you are using Monitor-166, the complete target application need to be stored in RAM space. This is required, since the Monitor changes the program code to set breakpoints in your application. Therefore the **ROM** entries in the dialog **Options – Target – External Memory** should refer to the RAM areas where you want to store the program code during debugging.

In the page **Options – L166 Misc** you should enter under **Reserve** the memory regions used by your Monitor configuration. The required address ranges for pre-configured Monitor variants can be found under **Options – Debug – Settings – Description**. For more information refer to “Monitor-166 Configuration” on page.



## Monitor-166 Configuration

The Monitor-166 can be adapted to different hardware configurations. The configuration works with a  $\mu$ Vision2 project file and is described in the file `C:\KEIL\C166\MONITOR\README.TXT`.

The following table describes the  $\mu$ Vision2 project files used for the Monitor configuration:

Folder and Project File	Description
<code>C:\KEIL\C166\MONITOR\USER166\USER 166.UV2</code>	User configuration files for 8xC166 CPU and ST10-166.
<code>C:\KEIL\C166\MONITOR\USER167\USER 167.UV2</code>	User configuration files for all other CPU variants, like 161/163/164/165 and 167 CPU type and ST10
<code>C:\KEIL\C166\MONITOR\...</code>	Other folders may contain pre-configured monitor variants for different boards.

To configure a new Monitor variant, create a new folder under `C:\KEIL\C166\MONITOR` and copy all files from `USER166` or `USER167`

## Trouble Shooting

If the Monitor does start correctly it is typically a problem of the CPU startup or Monitor code and data locations. Check carefully the settings in the file `CONFIG.INC`. In `BOOTSTRAP` mode,  $\mu$ Vision2 check if the Monitor is downloaded correctly. If the Monitor does not start, check if the Write Configuration Control bit `_WRCFG` matches your hardware.

If the Monitor stops working during the C startup code of your application you are most likely using different settings in the CPU parameter section. Check carefully if the settings of the `STARTUP.A66` or `START167.A66` file match the settings in the `CONFIG.INC`.

During operation the Monitor might report the following errors:

Error Text	Description
<code>BAD SERIAL TRANSMISSION</code>	$\mu$ Vision2 has lost the serial connection to the Monitor program. This error might occur because your program re-initializes the serial interface or changes the PORT direction register for I/O lines used by the serial interface. This error also occurs when you single step in the serial I/O routines of your application.
<code>CANNOT WRITE TO ROM AREA</code>	You try to download code into ROM space or non-existing memory.
<code>CANNOT WRITE BREAKPOINT</code>	You try to set a breakpoint in ROM space or non-existing memory.

Error Text	Description
CANNOT WRITE BREAKPOINT VECTOR	The Monitor program cannot install the interrupt vectors for the NMI trap or Serial interface. This error occurs when the RAM at address 0 cannot be accessed.
INVALID OPCODE	You try to execute invalid program code.





## Chapter 12. Command Reference

This chapter briefly describes the commands and directives for the Keil 166/ST10 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.*

---

11

### 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 [projectfile] [command]
```

*projectfile* is the name of a project file. µVision2 project files have the extension .UV2. If no *projectfile* is specified, µVision2 opens the last project file used.

*command* is one of the following commands. If no command is specified, µVision2 opens the *projectfile* in interactive **Build Mode**.

Command	Description
<b>-b</b>	Build the project and exit after the build process is complete.
<b>-d</b>	Start µVision2 Debugging Mode. You can use this command together with a <b>Debug Initialization File</b> to execute automated test procedures. µVision2 will exit after debugging is completed with the <b>EXIT</b> command or stop debug session.
<b>-r</b>	Re-translate the project and exit after the build process is complete.
<b>-t targetname</b>	Open the project and set the specified target as current target. This option can be used in combination with other µVision2 commands. Example: UV2 PROJECT1.UV2 -t "C167CR Board" -b builds the target "C167CR Board" as defined in the PROJECT1.UV2 file. If the <b>-t</b> command option is not given, µVision2 uses the <b>target</b> which was set as current target in the last project session.
<b>-o outputfile</b>	copy output of the <b>Output Window – Build</b> page to the specified file. Example: UV2 PROJECT1.UV2 -o "listmake.prn" -r

## A166 Macro Assembler Directives

### Invocation:

```
A166 sourcefile [directives]
A166 @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.

A166 Directive	Description
<b>CASE</b>	Enable case sensitive symbol names.
<b>COND, NOCOND</b>	Enable or disable skipped sections to appear in the listing file.
<b>DATE(date)</b>	Places <b>date</b> string in header (9 characters maximum).
<b>DEBUG</b>	Includes debugging symbol information in the object file.
<b>ERRORPRINT(filename)</b>	Outputs error messages to <b>filename</b> .
<b>EXPDECNUM</b>	Set the output format of the macro processor function %EVAL to decimal format.
<b>EXTMAC</b>	Enable extended ST10 MAC instructions.
<b>INCLUDE(filename)</b>	Includes the contents of <b>filename</b> in the assembly.
<b>GEN</b>	Generates a full listing of macro expansions in listing file.
<b>NOGEN</b>	List only the original source text in listing file.
<b>NOLINES</b>	Excludes line number information from the object file.
<b>NOLIST</b>	Excludes the assembler source code from the listing file.
<b>NOMPL</b>	Disables MPL macro processing.
<b>NOMACRO</b>	Disables standard macro processing.
<b>NOMOD166</b>	Do not recognize the predefined special function registers.
<b>MOD167</b>	Enable the extended instruction set for 161/163/164/165/167.
<b>NOSYMBOLS</b>	Excludes the symbol table from the listing file.
<b>NOSYMLIST</b>	Do not list the following symbol definitions in the symbol table.
<b>OBJECT(filename), NOOBJECT</b>	Enables or disables object file output. The object file is saved as <b>filename</b> if specified.
<b>PAGELength(n)</b>	Sets maximum number of lines in each page of listing file.
<b>PAGEWIDTH(n)</b>	Sets maximum number of characters in each line of listing file.
<b>PRINT(filename), NOPRINT</b>	Enables or disables listing file output. The listing file is saved as <b>filename</b> if specified.
<b>REGUSE</b>	Defines register usage of assembler functions for the C optimizer.
<b>RESET(symbol, ...)</b>	Assigns a value of 0000h to the specified symbols.
<b>RESTORE</b>	Restore control setting from SAVE stack.
<b>SAVE</b>	Stores current control setting for GEN, LIST and SYMLIST.
<b>SEGMENTED, NOSEGMENTED</b>	Define the mode of CPU operation.

A166 Directive	Description
<b>SET</b> ( <i>symbol, ...</i> )	Assigns a value of 0FFFFh to the specified symbols.
<b>TABS</b> ( <i>number</i> )	Specifies the tab setting.
<b>TITLE</b> ( <i>title</i> )	Includes <i>title</i> in the listing file header.
<b>TYPE, NOTYPE</b>	Defines whether type information is included in object file or not.
<b>USEDEXTONLY</b>	Prevent A166 from generating external definitions for unused external identifiers.
<b>XREF</b>	Includes a symbol cross reference report in the listing file.

## C166 Optimizing C Cross Compiler Directives

**Invocation:** `C166 sourcefile [directives]`  
`C166 @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**.

**directives** are control parameters described in the following table.

C166 Directive	Description
<b>ASM / ENDASM</b>	Merge assembler source text into the SRC file.
<b>ASMEXPAND, NOASMEXPAND</b>	Enable or disable macro text expansion for assembler source text sections.
<b>BROWSE</b>	Generate browse information.
<b>BYTEALIGN</b>	Assume pointers to byte-aligned structures.
<b>CODE</b>	Includes an assembly listing in the listing file.
<b>COMPACT</b>	Selects the <b>COMPACT</b> memory model.
<b>DEBUG</b>	Includes debugging information in the object file.
<b>DEFINE</b>	Defines preprocessor names on the command line.
<b>DYNAMICUSRSTK</b>	Enables dynamic modification of the user stack by a real-time OS.
<b>EXTJNS</b>	Break EXT instruction sequences at C source line numbers.
<b>FIXxxx</b>	Generate code with fixes for chip bugs.
<b>FLOAT64</b>	Enabled double-precision floating-point numbers.
<b>HCOMPACT</b>	Selects the <b>HCOMPACT</b> memory model.
<b>HLARGE</b>	Selects the <b>HLARGE</b> memory model.
<b>HOLD</b>	Specifies size limits for default placing of objects without explicit memory types.
<b>INCDIR</b>	Specify additional path names for include files.
<b>INIT, NOINIT</b>	Enable or disabled zero initialization of variables.
<b>LARGE</b>	Selects the <b>LARGE</b> memory model.
<b>LISTINCLUDE</b>	Includes the contents of include files in the listing file.
<b>MEDIUM</b>	Selects the <b>MEDIUM</b> memory model.
<b>MOD167</b>	Enable the extended instruction set for 161/163/164/165/167.
<b>NOALIAS</b>	Disable alias checking for pointer access optimization.
<b>NODPPSAVE</b>	Do not save the <b>DPP</b> registers in interrupt functions.
<b>NOCOND</b>	Excludes skipped conditional code from the listing file.
<b>NOEXTEND</b>	Disables 166 extensions and processes only ANSI C constructs.
<b>NOFIXDPP</b>	Generate code without <b>DPP</b> register assumptions.
<b>NOFRAME</b>	Suppress prolog and epilog for interrupt service routines

C166 Directive	Description
<b><u>OBJECT</u></b> [( <i>filename</i> )], <b><u>NOOBJECT</u></b>	Enables or disables object file output. The object file is saved as <i>filename</i> if specified.
<b><u>OPTIMIZE</u></b>	Specifies the level of optimization performed by the compiler.
<b><u>ORDER</u></b>	Locates variables in memory in the same order in which they are declared in the source file.
<b><u>PACK</u></b>	Generate BYTE aligned structures with word elements.
<b><u>PAGELLENGTH</u></b> ( <i>n</i> )	Sets maximum number of lines in each page of listing file.
<b><u>PAGEWIDTH</u></b> ( <i>n</i> )	Sets maximum number of characters in each line of listing file.
<b><u>PECDEF</u></b>	Reserve <b>PEC</b> channels.
<b><u>PREPRINT</u></b> [( <i>filename</i> )]	Produce a preprocessor listing file with all macros expanded. The preprocessor listing file is saved as <i>filename</i> if specified.
<b><u>PREPRINTONLY</u></b> [( <i>file</i> )]	Produce a preprocessor listing and stop compilation.
<b><u>PRINT</u></b> [( <i>filename</i> )], <b><u>NOPRINT</u></b>	Enables or disables listing file output. The listing file is saved as <i>filename</i> if specified.
<b><u>REGFILE</u></b> ( <i>filename</i> )	Specifies the name of the generated file to contain register usage information.
<b><u>RENAMECLASS</u></b>	Rename predefined class names in the object file.
<b><u>SAVESYS</u></b>	Save temporary results and variables to system stack.
<b><u>SAVEUSR</u></b>	Save temporary results and saved-by-callee variables to user stack.
<b><u>SMALL</u></b>	Selects the <b>SMALL</b> memory model.
<b><u>SRC</u></b>	Creates an assembly source file instead of an object file.
<b><u>STATIC</u></b>	Allocate automatic variables to static memory addresses.
<b><u>SYMBOLS</u></b>	Includes a list of the symbols used in the listing file.
<b><u>TINY</u></b>	Selects the <b>TINY</b> memory model.
<b><u>UNSINGEDCHAR</u></b>	Treat plain <b>char</b> as <b>unsigned char</b> .
<b><u>USERSTACKDPP3</u></b>	Assume user stack area in <b>IDATA</b> or <b>SDATA</b> memory class.
<b><u>WARNING</u></b>	Change a Warning to an Error or disable a warning.
<b><u>WARNINGLEVEL</u></b> ( <i>n</i> )	Controls the types and severity of warnings generated.

## L166 Linker/Locator Directives

The L166 Linker/Locator links your 166 object modules, locates them at absolute addresses, and creates an absolute object module you may use for debugging or creating an Intel HEX file. Invoke the linker using either of the following command lines:

```
L166 @commandfile
L166 inputlist [TO outputfile] [directives]
```

where

- inputlist** is a comma-separated list of the object files and libraries the linker includes in the final absolute object module.
- 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. The command file includes an **inputlist**, **outputfile** (if desired), and **directives**. You may use a command file to make linking your application easier or to include more input files or directives than fit on the command line.
- directives** are linker control parameters described in the following table.

L166 Directive	Description
<b>ASSIGN</b>	Define public symbols on the command line.
<b>CINITTAB</b>	Locate C initialization data sections to specified address range.
<b>CLASSES</b>	Define physical class address ranges and class locating order.
<b>DISABLEWARNING</b>	Disable report of specified warning numbers.
<b>DPPUSE</b>	Re-assign DPP registers for <b>NCONST</b> and <b>NDATA</b> groups.
<b>GROUPS</b>	Define physical group addresses and group locating order.
<b>IXREF</b>	Include a cross reference report in the listing file.
<b>LINKONLY</b>	Suppress located process for incremental linkage.
<b>NAME</b>	Specifies a module name for the object file.
<b>NOCASE</b>	Disable case sensitivity of the linker.
<b>NOCOMMENTS</b>	Excludes comment information from the listing file and the object file.
<b>NODEFAULTLIBRARY</b>	Excludes modules from the run-time libraries.
<b>NOLINES</b>	Excludes line number information from the listing file and object file.
<b>NOMAP</b>	Excludes memory map information from the listing file.
<b>NOPUBLICS</b>	Excludes public symbol information in listing and object file.
<b>NOSYMBOLS</b>	Excludes local symbol information in listing and object file.
<b>NOTYPES</b>	Excludes type information from the object file.
<b>NOVECTAB</b>	Remove the interrupt vector table in the output file.

L166 Directive	Description
<b><u>OBJECT</u>CONTROLS</b>	Excludes specific debugging information from the object file. Subcontrols must be specified in parentheses. See <b>NO</b> COMMENTS, <b>NO</b> LINES, <b>NO</b> PUBLICS, <b>NO</b> SYMBOLS, and <b>PURGE</b> .
<b><u>PAGE</u>LENGTH(<i>n</i>)</b>	Sets maximum number of lines in each page of listing file.
<b><u>PAGE</u>WIDTH(<i>n</i>)</b>	Sets maximum number of characters in each line of listing file.
<b><u>PRINT</u></b>	Specifies the name of the listing file.
<b><u>PRINT</u>CONTROLS</b>	Excludes specific debugging information from the listing file. Subcontrols must be specified in parentheses. See <b>NO</b> COMMENTS, <b>NO</b> LINES, <b>NO</b> PUBLICS, <b>NO</b> SYMBOLS, and <b>PURGE</b> .
<b><u>PURGE</u></b>	Excludes all debugging information from the listing file and the object file.
<b><u>REG</u>BANK</b>	Define physical register bank addresses and locating order.
<b><u>REG</u>FILE(<i>filename</i>)</b>	Specifies the name of the register usage information file.
<b><u>RESERVE</u></b>	Reserve physical 166/ST10 memory areas.
<b>RTX166</b>	Includes support for the RTX166 Full real-time kernel.
<b>RTX166TINY</b>	Includes support for the RTX166 Tiny real-time kernel.
<b><u>SECS</u>IZE</b>	Change the length of sections.
<b><u>SECTIONS</u></b>	Define physical section addresses and section locating order.
<b><u>VECT</u>AB</b>	Specify a start address for the interrupt vector table.
<b><u>WARNING</u>LEVEL(<i>n</i>)</b>	Controls the types and severity of warnings generated.

## LIB166 Library Manager Commands

The LIB166 Library Manager lets you create and maintain library files of your 166 object modules. Invoke the library manager using the following command:

```
LIB166 [command]
LIB166 @commandfile
```

*command* is one of the following commands. If no command is specified LIB166 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 LIB166. You may use a command file to generate a large library with at once.

LIB166 Command	Description
<b><u>A</u>DD</b>	Adds an object module to the library file. For example, LIB166 ADD GOODCODE.OBJ TO MYLIB.LIB adds the <b>GOODCODE.OBJ</b> object module to <b>MYLIB.LIB</b> .
<b><u>C</u>REATE</b>	Creates a new library file. For example, LIB166 CREATE MYLIB.LIB creates a new library file named <b>MYLIB.LIB</b> .
<b><u>D</u>ELETE</b>	Removes an object module from the library file. For example, LIB166 DELETE MYLIB.LIB (GOODCODE) removes the <b>GOODCODE</b> module from <b>MYLIB.LIB</b> .
<b><u>E</u>XTRACT</b>	Extracts an object module from the library file. For example, LIB166 EXTRACT MYLIB.LIB (GOODCODE) TO GOOD.OBJ copies the <b>GOODCODE</b> module to the object file <b>GOOD.OBJ</b> .
<b><u>E</u>XIT</b>	Exits the library manager interactive mode.
<b><u>H</u>ELP</b>	Displays help information for the library manager.
<b><u>L</u>IST</b>	Lists the module and public symbol information stored in the library file. For example, LIB166 LIST MYLIB.LIB TO MYLIB.LST PUBLICS generates a listing file (named <b>MYLIB.LST</b> ) that contains the module names stored in the <b>MYLIB.LIB</b> library file. The <b>PUBLICS</b> directive specifies that public symbols are also included in the listing.
<b><u>R</u>EPLACE</b>	Replaces an existing object module to the library file. For example, LIB166 REPLACE GOODCODE.OBJ IN MYLIB.LIB replaces the <b>GOODCODE.OBJ</b> object module in <b>MYLIB.LIB</b> . Note that Replace will add <b>GOODCODE.OBJ</b> to the library if it does not exist.
<b><u>T</u>RANSFER</b>	Generates a complete new library and adds object modules. For example, LIB166 TRANSFER FILE1.OBJ, FILE2.OBJ TO MYLIB.LIB deletes the existing library <b>MYLIB.LIB</b> , re-creates it and adds the object modules <b>FILE1.OBJ</b> and <b>FILE2.OBJ</b> to that library.



## OH166 Object-HEX Converter Commands

The OH166 object-HEX converter creates Intel HEX files from absolute object modules. Invoke the HEX converter using the following command:

```
OH166 absfile [H167] [RANGE(start-end)] [OFFSET(offset)] [FLASH(fillbyte)]
```

where

**absfile** is the name of an absolute object file that the L166 linker/locator generated.

**H167** specifies that an Intel HEX-386 file is created. By default, OH166 creates a standard Intel HEX-86 file.

**RANGE** specifies the address range of data in the *absfile* to convert and store in the HEX file. The default range is 0x000000 to 0xFFFFFFFF.

**start** specifies the starting address of the range. This address must be entered in C hexadecimal notation, for example: 0x010000.

**end** specifies the ending address of the range. This address must be entered in C hexadecimal notation, for example: 0xFFFFFFFF.

**OFFSET** specifies an offset (*offset*) added to the addresses from the *absfile*.

**FLASH** The HEX file is sorted in ascending order. Unused bytes in the **RANGE** are filled with the *fillbyte* specified with the FLASH directive. The sorted HEX file can be downloaded to FLASH devices.

The following command line creates a HEX file named **MYCODE.HEX** from the absolute object module **MYCODE**. Records in the HEX file are written in HEX-386 format.

```
OH166 MYCODE H167
```

# Index

## \$

\$ system variable 85

—

\_break\_ system variable 85

\_getkey\_ library routine 142

\_idle\_ library routine 150

\_pwrnd\_ library routine 150

\_sof\_ library routine 135

\_srvwdt\_ library routine 145

## μ

μVision2 Debugger 67

μVision2 IDE 6,13,36

    Command line parameters 163

    Debug Options 75

    Menu Commands 14

    Options 40

    Shortcuts 14

    Toolbars 14

    Toolbox 74

## 1

166 Devices 2

166 microcontroller family 2

## A

A/D converter 149

    Example program 149

A166 8

A166 macro assembler 28

    Commands 164

    Directives 164

**Access Break** 70

ADCIC register 149

ADCON register 149

Add command

    LIB166 library manager 170

ADDAT register 149

ADDAT2 register 149

Additional items, document

    conventions iv

ADEIC register 149

Analog/Digital converter 149

ANSI C 110

Assembler Instructions 68

Assembler kit 8

Assign directive

    L166 linker/locator 168

Assistance 5

## B

Binary constants 83

Bit addresses 94

Bit-fields 49

Bold capital text, use of iv

Bootstrap loader 156

Braces, use of iv

break 97

Breakpoint Commands 81

Breakpoints 69

    Access Break 70

    Conditional 70

    Execution Break 70

Build Process 57

Build Project 41

## C

C run-time configuration 154

C startup code 153

C166 C compiler 20

C166 compiler

    Commands 166

    Directives 166

    Language Extensions 20

    Memory Models 23

CA166 8

CAPREL register 140

case 97

Changes to the documentation 3

Character constant escape

    sequence 84

Character constants	84	Create a Library	59
Choices, document conventions	iv	Create a Project File	36
Classes directive		Create command	
L166 linker/locator	168	LIB166 library manager	170
Classes of symbols	91	Create HEX File	41
Code Coverage	77	Creating header files	128
Command reference	163	CRIC register	140
A166 macro assembler	164	Custom Translator	64
C166 compiler	166	cycles system variable	85
L166 linker/locator	168		
LIB166 library manager	170	<b>D</b>	
OH166 hex converter	171		
Comparison chart	9	Data Threshold	50
Compiler kit	8	Data Types	54
conditional assembly	28	DAVE utility	154
<b>Conditional Break</b>	70	Debug Comamnds	
Configuration		Program Execution	80
C run-time	154	Debug Commands	17,80
CPU	153	Memory	80
DAVE utility	154	Debug Functions	97
Hardware	154	Debug Menu	17,68,72
Memory model	153	Debug Mode	67
Run-time	154	Debugger	67
tool options	43	Decimal constants	83
Constant Expressions	82	Delete command	
Constants	82	LIB166 library manager	170
Binary	83	Development cycle	6
Character	84	Development tools	13
Decimal	83	Device Database	58
Floating-point	83	Direct memory access	
HEX	83	controllers	134
Octal	83	Directives	
String	84	A166 macro assembler	164
continue	97	C166 compiler	166
Control Directives		L166 linker/locator	168
#pragma	26	LIB166 library manager	170
Copy Tool Settings	60	OH166 hex converter	171
Correct syntax errors	41	Directory structure	12
Counters	140	Disassembly Window	68
Courier typeface, use of	iv	Displayed text, document	
CPU driver symbols	85	conventions	iv
CPU initialization	153	DMA	134
CPU pin register	<i>See</i> VTREG	do	97
CPU registers	73	Document conventions	iv
CPU Registers	73	Documentation changes	3
CPU Simulation	67	Double brackets, use of	iv

DPP registers 129  
dScope functions 110

## E

Edit Menu 15  
Editor Commands 15  
EK166 evaluation kit 4  
Ellipses, use of iv  
Ellipses, vertical, use of iv  
else 97  
Escape sequence 84  
Evaluation board 155  
Evaluation kit 4  
Evaluation users 4  
Example program  
    A/D converter 149  
    General purpose timers 140  
    Idle mode 150  
    Interrupt functions 131  
    Parallel port I/O 139  
    PEC 137  
    Peripheral event controller 137  
    Power down mode 150  
    Pulse width modulation 147  
    PWM 147  
    Serial interface 142  
    Watchdog timer 145  
Examples of expressions 95  
exec  
    Function description 100  
    Predefined function 100  
**Execution Break** 70  
Exit command  
    LIB166 library manager 170  
Experienced users 4  
Expression components 82  
    Bit addresses 82,94  
    Constants 82  
    CPU driver symbols 85  
    Line numbers 82,93  
    Operators 82,94  
    Program variables 82,90  
    SFRs 85  
    Special function registers 85  
    Symbols 82,90  
    System variables 82,85

Type specifications 82,94  
Variables 82,90  
VTREGs 86  
Expression examples 95  
Expressions 82  
External RAM 55,59  
Extract command  
    LIB166 library manager 170

## F

Feature check list 9  
File Commands 14  
File Extensions 65  
File Menu 14  
File specific Options 43,62  
Filename, document conventions iv  
Files Groups 42  
Find in Files 44  
float  
    Predefined function 100  
Floating-point constants 83  
Folder for Listing Files 58  
Folder for Object Files 58  
FR166 9  
Fully qualified symbols 91  
Function classes  
    Predefined functions 99  
    Signal functions 99  
    User functions 99  
Function Classes 99  
Functions  
     $\mu$ Vision2 Debug Functions 97  
    Classes 99  
    Creating 97  
    Invoking 99  
    Predefined 100  
    Signal 106  
    User 104

## G

General commands 81  
General purpose timers 140  
    Example program 140  
getfloat

Function description	101	Layout	132
Predefined function	100	Layout for PEC	136
getint		Interrupt enable flag	133
Predefined function	100	Interrupt functions	131
getlong		Example program	131
Predefined function	100	Interrupt group level	133
Getting help	5	Interrupt priority level	133
Getting started immediately	3	Interrupt request flag	133
Global Register Optimization	50	Introduction	1
goto	97	Italicized text, use of	iv
GPT1	140	itrace system variable	85
GPT2	140		
Group specific Options for Groups	43,62		
		<b>K</b>	
<b>H</b>		Key names, document conventions	iv
Hardware configuration	154	Kit comparison	9
Hardware requirements	11		
HCOMPACT Memory Model	49	<b>L</b>	
Header files	128	L166 linker/locator	30
Help	5	Assign directive	168
Help command		Classes directive	168
LIB166 library manager	170	Commands	168
Help Menu	18	Directives	168
HEX constants	83	Ixref directive	168
HEX File	41	Name directive	168
HLARGE Memory Model	49	Nocomments directive	168
		Nodefaultlibrary directive	168
<b>I</b>		Last minute changes	3
I/O ports	88,138	LIB166 library manager	34
IDLE instruction	150	Add command	170
Idle mode	150	Commands	170
Example program	150	Create command	170
IEN register	132,150	Delete command	170
if97		Exit command	170
Import $\mu$ Vision1 Projects	56	Extract command	170
Include specific Library		Help command	170
Modules	64	List command	170
Installing the software	11	Replace command	170
Instructions		Transfer mmand	170
IDLE	150	Library	63,64
PWRDN	150	Line numbers	93
RETI	150	List command	
SRVWDT	145	LIB166 library manager	170
Interrupt control registers	132	Listing Files	58
		Literal	91

Literal symbols	91	OH166 hex converter	34
Locate Sections	43,60	Command-line example	171
		Commands	171
<b>M</b>		Omitted text, document	
Macro Processing Language	28	conventions	iv
Macros		Operators	94
standard macros	28	Optimum Code	48
Manual topics	3	Optional items, document	
MEDIUM Memory Model	48	conventions	iv
Memory Map	78	Options	
Memory model	153	for Files	43,62
Memory Model	40,48	for Groups	43,62
COMPACT	23	<b>Output Window</b>	41
HCOMPACT	23		
HLARGE	23	<b>P</b>	
LARGE	23	Parallel port	138
MEDIUM	23	Example program	139
SMALL	23	Part numbers	8
TINY	23	PC-Lint	47
Memory Models	23	PEC	134
Memory Type	40,48	Control register layout	135
Memory Types	22	Example program	137
Memory Window	73	Interrupt control register	
memset		layout	136
Function description	101	Performance Analyzer	77
Predefined function	100	Peripheral event controller	134
Module names	90	Control register layout	135
Monitor Driver	157	Example program	137
Monitor-166	155	Interrupt control register	
Configuration	160	layout	136
Serial Transmission Line	156	PK161	8
MPL	28	PK166	8
		Pointer	23
<b>N</b>		Port I/O	138
Naming conventions for		Ports	88
symbols	90	Power down mode	150
<b>New Project</b>	36	Example program	150
New users	4	Power reduction modes	150
NMI pin	150	PPx register	146
Non-qualified symbols	92	Predefined functions	100
		exec	100
<b>O</b>		float	100
Object Files	58	getfloat	100,101
Octal constants	83	getint	100
		getlong	100

memset	100,101	ADDAT2	149
printf	100,101	ADEIC	149
rand	100,102	CAPREL	140
twatch	100,102	CRIC	140
Printed text, document		IEN	132,150
conventions	iv	PP <sub>x</sub>	146
printf		PSW	132
Function description	101	PT <sub>x</sub>	146
Predefined function	100	PWMCON0	146
printf library routine	142	PWMCON1	146
Product link	8	PWMIC	146
Production kit	4	PW <sub>x</sub>	146
Professional Developer's Kit	8	S0BG	142
Program counter system variable	85	S0CON	142
Project Commands	17	S0EIC	142
Project Menu	17	S0RBUF	142
<b>Project Targets</b>	42	S0RIC	142
PSW register	132	S0TBIC	142
PT <sub>x</sub> register	146	S0TBUF	142
Pulse width modulation	146	S0TIC	142
Example program	147	T2	140
putchar library routine	142	T2CON	140
PWM	146	T2IC	140
Example program	147	T3	140
PWMCON0 register	146	T3CON	140
PWMCON1 register	146	T3IC	140
PWMIC register	146	T4	140
PWRDN instruction	150	T4CON	140
PW <sub>x</sub> register	146	T4IC	140
		T5	140
		T5CON	140
		T5IC	140
		T6	140
		T6CON	140
		T6IC	140
		WDTCON	145
		Replace command	
		LIB166 library manager	170
		Requesting assistance	5
		Requirements	11
		RETI instruction	150
		RS-232	142
		RS-232 ports	89
		RTX166	9
		Run Program	72
<b>Q</b>			
Qualified symbols	91		
<b>R</b>			
radix system variable	85		
rand			
Function description	102		
Predefined function	100		
Real-time operating system	9		
REG167.H header file	128		
Register banks	131		
Registers			
ADCIC	149		
ADCON	149		
ADDAT	149		

**S**

S0BG register	142
S0CON register	142
S0EIC register	142
S0RBUF register	142
S0RIC register	142
S0TBIC register	142
S0TBUF register	142
S0TIC register	142
Sans serif typeface, use of	iv
Saving power	150
scanf library routine	142
Select Text	16
Serial interface	142
Example program	142
Serial ports	89
Serial Window	76
SETUP program	11
SFRs	85
Siemens	
DAVE utility	154
Signal functions	106
Simulating I/O ports	88
Simulating serial ports	89
Simulation	67
Simulator	67
Single Step Program	72
Single-board computers	155
SMALL Memory Model	48
Software development cycle	6
Software requirements	11
Source Browser	44
Special function registers	85
SRVWDT instruction	145
ST10 Devices	2
Start $\mu$ Vision2	36
Start Debugging	67
Start External Tools	57
START167.A66	38,153
START167.A66 file	145
Startup code	128,145,153
Memory model	153
STARTUP.A66	153
String constants	84
switch	97

Symbol expressions	90
Symbols	
Classification	91
CPU driver	85
Fully qualified	91
Literals	91
Module names	90
Naming conventions	90
Non-qualified	92
Qualified names	91
SFRs	85
Special function registers	85
System variables	85
VTREGs	86
Symbols Window	79
Syntax errors	41
System variables	85
\$ 85	
_break_	85
cycles	85
itrace	85
Program counter	85
radix	85

**T**

T2 register	140
T2CON register	140
T2IC register	140
T3 register	140
T3CON register	140
T3IC register	140
T4 register	140
T4CON register	140
T4IC register	140
T5 register	140
T5CON register	140
T5IC register	140
T6 register	140
T6CON register	140
T6IC register	140
Target hardware	155
Target Tool Settings	60
Technical support	5
Timer 2	140
Timer 3	140
Timer 4	140



Timer 5	140
Timer 6	140
Timers	140
Tool Information	65
tool options	43
Tools Menu	18,46
Topics	3
Transfer command	
LIB166 library manager	170
Translate asm/endasm sections	63
twatch	
Function description	102
Predefined function	100
Type specifications	94
Types of users	4

## U

---

UART	76
User Classes	61
User functions	104
Users	4
Using Monitor	155
Utilities	44

## V

---

V <sub>AGND</sub> pin	149
V <sub>AREF</sub> pin	149
Variable expressions	90
Variable values	72
Variables, document	
conventions	iv
Vertical bar, use of	iv
Vertical ellipses, use of	iv
View memory contents	73
View Menu	16
VTREGs	86

## W

---

Watch Window	72
Watchdog timer	145
Example program	145
WDTCN register	145
while	97
Window Menu	18
Windows-based tool	
requirements	11
Write Optimum Code	48