



# **8051/251 Evaluation Kit**

**Getting Started with the 8051 and MCS<sup>®</sup> 251  
Microcontroller Development Tools**

**User's Guide 11.97**

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 1990-1998 Keil Elektronik GmbH and Keil Software, Inc.  
All rights reserved.

Keil C51™ and dScope™ are trademarks of Keil Elektronik GmbH.  
Microsoft®, MS-DOS®, and Windows™ are trademarks or registered trademarks of Microsoft Corporation.  
IBM®, PC®, and PS/2® are registered trademarks of International Business Machines Corporation.  
Intel®, MCS® 51, MCS® 251, ASM-51®, and PL/M-51® are registered trademarks of Intel Corporation.

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 8051 and MCS<sup>®</sup> 251 microcontroller software development tools. 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.

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

“Chapter 3. 8051/251 Product Line” discusses the different products that we offer for the 8051 and 251 microcontrollers. Read this chapter to determine which product provides the tools you need.

“Chapter 4. 8051 Development Tools” describes the major features of our 8051 development tools including the C compiler, assembler, debugger, and integrated development environment.

“Chapter 5. 251 Development Tools” describes the major features of our 251 development tools including the C compiler, assembler, debugger, and integrated development environment.

“Chapter 6. Using the 8051/251 tools” describes the provided sample programs along with a step-by-step guide that shows how to build them using our tools.

“Chapter 7. Hardware Products” introduces our hardware-based tools that you can use to aid in development and debugging. Our evaluation boards for the 80C517A and 80C251SB and our EPROM emulator are discussed.

“Chapter 8. Real-Time Kernels” discusses the RTX-51 Tiny and RTX-51 Full real-time operating systems. This chapter provides an overview of multitasking systems, why they are desirable, and how they are used.

“Chapter 9. Command Reference” briefly describes the commands and controls for our 8051 and 251 development tools.

---

## **NOTE**

*This manual assumes that you are familiar with Microsoft Windows and the hardware and instruction set of the 8051 and MCS<sup>®</sup> 251 microcontrollers.*

---

# 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 MS-DOS command prompt. This text usually represents commands that you must type in literally. For example:  <div style="text-align: center;"> <b>CLS                    DIR                    BL51.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>void main (void) { . . . while (1);</pre>
[ <i>Optional Items</i> ]	Optional arguments in command-line and option fields are indicated by double brackets. For example:  <pre>C51 TEST.C PRINT [ (filename) ]</pre>
{ <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>
Manual Topics .....	1
Evaluation and Demo Kits .....	2
Types of Users .....	2
Changes to the Documentation .....	3
Requesting Assistance.....	3
<b>Chapter 2. Installation.....</b>	<b>7</b>
System Requirements.....	7
Backing Up Your Disks .....	8
Installing the Software .....	8
Directory Structure .....	9
Environment Settings.....	10
Improving System Performance.....	11
<b>Chapter 3. 8051/251 Product Line.....</b>	<b>13</b>
8051 Development Tool Kits.....	13
251 Development Tool Kits.....	17
Subscription Kits .....	20
Tool Kit Comparison Chart .....	22
<b>Chapter 4. 8051 Development Tools.....</b>	<b>23</b>
8051 Microcontroller Family.....	23
C51 Optimizing C Cross Compiler .....	24
A51 Macro Assembler.....	41
BL51 Code Banking Linker/Locator .....	43
OC51 Banked Object File Converter .....	47
OH51 Object-Hex Converter .....	47
LIB51 Library Manager.....	47
dScope-51 for Windows .....	47
$\mu$ Vision/51 for Windows .....	48
<b>Chapter 5. 251 Development Tools.....</b>	<b>51</b>
MCS <sup>®</sup> 251 Microcontroller Family.....	51
C251 Optimizing C Cross Compiler .....	52
A251 Macro Assembler .....	58
L251 Code Banking Linker/Locator .....	59
OH251 Object-Hex Converter .....	60
LIB251 Library Manager.....	60
dScope-251 for Windows .....	60
$\mu$ Vision/251 for Windows .....	61
<b>Chapter 6. Using the 8051/251 tools .....</b>	<b>63</b>
Starting $\mu$ Vision and dScope .....	64
$\mu$ Vision IDE Overview .....	64
dScope Simulator/Debugger Overview.....	70

---

Sample Programs .....	79
HELLO: Your First 8051/251 C Program.....	81
MEASURE: A Remote Measurement System .....	88
BADCODE: An Example with Syntax Errors.....	105
<b>Chapter 7. Hardware Products .....</b>	<b>107</b>
ProROM EPROM Emulator .....	107
MCB517A Evaluation Board.....	108
MCB251SB Evaluation Board.....	109
<b>Chapter 8. Real-Time Kernels .....</b>	<b>111</b>
RTX-51 Real-Time Operating System.....	111
<b>Chapter 9. Command Reference .....</b>	<b>121</b>
A51/A251 Macro Assemblers.....	121
C51/C251 Compiler.....	122
L51/BL51 Linker/Locator.....	124
L251 Linker/Locator.....	126
OC51 Banked Object File Converter.....	127
OH51 Object-Hex Converter.....	127
OH251 Object-Hex Converter.....	128
LIB51/LIB251 Library Manager.....	128
<b>Index.....</b>	<b>129</b>

# Chapter 1. Introduction

# 1

Thank you for allowing Keil Software to provide you with software development tools for the 8051 and 251 family of microprocessors. With our tools, you can generate embedded applications for the multitude of 8051 and 251 derivatives. Our 8051 and 251 development tools are listed below:

- C51/C251 Optimizing C Cross Compiler,
- A51/A251 Macro Assembler,
- 8051/251 Utilities (linker, object file converter, library manager),
- dScope for Windows™ Source-Level Debugger/Simulator,
- $\mu$ Vision for Windows™ Integrated Development Environment.

These tools are combined into the kits described in “Chapter 3. 8051/251 Product Line” on page 13. The individual tools are described in detail in “Chapter 4. 8051 Development Tools” on page 23.

In addition to the above development tools, we also provide real-time kernels, evaluation boards, and debugging hardware. Refer to “Chapter 8. Real-Time Kernels” on page 111 and “Chapter 7. Hardware Products” on page 107 for more information about these products.

Our tools are designed for the professional software developer, but any level of programmer can use them to get the most out of the 8051 and 251 hardware.

## Manual Topics

This manual discusses a number of topics including how to:

- Install the software on your system (see “Chapter 2. Installation” on page 7) and fine tune it for maximum performance (see “Improving System Performance” on page 11),
- Select the best tool kit for your application (see “Chapter 3. 8051/251 Product Line” on page 13),
- Use the 8051 development tools (see “Chapter 4. 8051 Development Tools” on page 23),

- Run the included sample programs (see “Chapter 6. Using the 8051/251 tools” on page 63).

If you want to get started immediately, you may do so by installing the software (refer to “Chapter 2. Installation” on page 7) and running the sample programs (refer to “Chapter 6. Using the 8051/251 tools” on page 63). This is all you need to do to begin using this kit.

## Evaluation and Demo Kits

Keil Software provides two kits that let you evaluate our tools.

The **8051/251 Demo Kit** includes demonstration versions of our tools. The tools in the Demo Kit do not generate actual object code. They generate listing files where you can see the code generated by the compiler and other tools.

The **8051/251 Evaluation Kit** includes evaluation versions of our tools. The tools in the Evaluation Kit let you generate applications up to 2 Kbytes in size. You may use this kit to evaluate the effectiveness of our tools and to generate small target applications.

Both kits include this user’s guide and software. This user’s guide is also included in each of our tool kits.

## 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 copies of the development tools. You may use the included sample programs to get real-world experience with our 8051 and 251 development tools. Even if you are only a 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 our 8051 development tools for the first time. The included software provides you with the latest development



tool versions as well as sample programs. If you are new to the 8051 or 251 or the tools, take the time to review the sample programs described in this manual. This manual provides a quick tutorial and helps new or inexperienced users quickly get started with the tools.

**Experienced Users** are those users who have previously used our 8051 development tools and are now upgrading to the latest 8051 or 251 tools. The software included with a product upgrade contains the latest development tools, the sample programs, and a full set of manuals.

## Changes to the Documentation

Last minute changes and corrections to the software and manuals are listed in the **README.TXT** file which is included in the root directory of your installation. Take the time to read this file to determine if there are any changes that may impact your installation.

## Requesting Assistance

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.
3. Isolate the problem to determine if it is a problem with the assembler, compiler, linker, library manager, or another development tool.
4. 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. If you contact us by fax, be sure to include your name, your product serial number and version number, and telephone numbers (voice and fax) where we can reach you.

**1**

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 fax it to us. However, please try not send long listings as this slows down our response.





## 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.

---

### **NOTE**

*This chapter refers to various MS-DOS commands which may be used to customize your operating environment. The **SET** and **PATH** commands, for example, are used to initialize environment variables used by the compiler and utilities. If you are not familiar with these commands and other MS-DOS operations mentioned in this chapter, please refer to your DOS user's guide.*

---

## 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:

- 100% IBM compatible 386 or higher PC,
- Windows 3.1 or higher,
- 4 MB RAM minimum,
- Hard disk with 6 MB free disk space.

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

- 100% IBM compatible 386 or higher PC with 640 KB RAM,
- MS-DOS Version 3.1 or higher,
- Hard disk with 6 MB free disk space.

The C compiler and utilities require that you have at least 20 files and 20 buffers defined in your `CONFIG.SYS` file. Additionally, you need enough environment space for the environment variables used by the compiler and utilities (see “Environment Settings” on page 10).

Your `CONFIG.SYS` file should look similar to the following:

```
BUFFERS=20
FILES=20
SHELL=C:\COMMAND.COM /e:1024 /p
```

If you receive the message `out of environment space` from DOS, you can increase the amount of environment space by increasing the number `1024` in the above example. Refer to your DOS user’s guide for more information.

## Backing Up Your Disks

We strongly suggest that you make a backup copy of the installation diskettes using the DOS `COPY` or `DISKCOPY` commands. Then, use the backup disks to install the software. Be sure to store the original disks in a safe place in case your backups are lost or damaged.

## Installing the Software

All of our products come with an installation program which allows easy installation of our software.

## Installing DOS-Based Products

To install DOS-based products, insert the first product diskette into Drive A and enter the following command line at the DOS prompt:

```
A: INSTALL
```

Then, follow the instructions displayed by the installation program.

## Installing Windows-Based Products

To install Windows-based products...

- Insert the first product diskette into Drive A,
- Select the **Run...** command from the **File** menu in the Program Manager,
- Enter **A:SETUP** at the **Command Line** prompt,
- Select the **OK** button.

Then, follow the instructions displayed by the installation program.

## Directory Structure

The installation program copies the development tools into subdirectories of the following base directories. The directory used depends on the kit being installed.

Directory	Description
\C51	8051 development tools.
\C51EVAL	8051 evaluation tools.
\C251	251 development tools.
\C251EVAL	251 evaluation tools.

After creating the appropriate directory, the installation program copies the development tools into the subdirectories listed in the following table.

Subdirectory	Description
...\ASM	Assembler include files.
...\BIN	Executable files.
...\DS51	dScope-51 for DOS IOF drivers.
...\EXAMPLES	Sample applications.
...\RTX51	RTX-51 Full files.
...\RTX_TINY	RTX-51 Tiny files.
...\INC	C compiler include files.
...\LIB	C compiler library files and startup code.
...\MON51	Target monitor files.
...\TS51	tScope-51 for DOS IOT drivers.

This table lists a complete installation that includes the entire line of 8051 development tools. Your installation may vary depending on the products you purchased.

## Environment Settings

# 2

The compiler and utilities require entries in the DOS environment table that specify the path to include files and libraries. In addition, you must include the `...\BIN\` directory in your `PATH`.

The following table lists the environment variables, their default paths, and a brief description.

Variable	Path	Description
<code>PATH</code>	<code>\C51\BIN</code>	Specifies the path of the 8051 development tools.
<code>PATH</code>	<code>\C51EVAL\BIN</code>	Specifies the path of the 8051 evaluation tools.
<code>PATH</code>	<code>\C251\BIN</code>	Specifies the path of the 251 development tools.
<code>PATH</code>	<code>\C251EVAL\BIN</code>	Specifies the path of the 251 evaluation tools.
<code>TMP</code>		Specifies the path for temporary files generated. For best performance, the path specified should be a RAM disk. If this environment variable is specified, the path must exist. If the path does not exist, the tools abort reporting a fatal error.
<code>C51INC</code>	<code>\C51\INC</code>	Specifies the path where the standard C51 compiler include files are located.
<code>C251INC</code>	<code>\C251\INC</code>	Specifies the path where the standard C251 compiler include files are located.
<code>C51LIB</code>	<code>\C51\LIB</code>	Specifies the path where the standard C51 compiler library files are located.
<code>C251LIB</code>	<code>\C251\LIB</code>	Specifies the path where the standard C251 compiler library files are located.

### NOTE

*This manual makes references to programs and files in the `\C51\...` directory. This directory is equivalent to the `\C51EVAL\...`, `\C251\...`, and `\C251EVAL\...` directories.*



Typically, environment settings are automatically installed in your **AUTOEXEC.BAT** file by the installation program. If you wish to put these settings in a separate batch file, the environment settings must be entered as follows:

8051 Development Tools	8051 Evaluation Tools
<code>PATH=C:\C51\BIN;...</code>	<code>PATH=C:\C51EVAL\BIN;...</code>
<code>SET C51INC=C:\C51\INC</code>	<code>SET C51INC=C:\C51EVAL\INC</code>
<code>SET C51LIB=C:\C51\LIB</code>	<code>SET C51LIB=C:\C51EVAL\LIB</code>

251 Development Tools	251 Evaluation Tools
<code>PATH=C:\C251\BIN;...</code>	<code>PATH=C:\C251EVAL\BIN;...</code>
<code>SET C251INC=C:\C251\INC</code>	<code>SET C251INC=C:\C251EVAL\INC</code>
<code>SET C251LIB=C:\C251\LIB</code>	<code>SET C251LIB=C:\C251EVAL\LIB</code>

## Improving System Performance

There are two methods you can employ to improve performance of the C51 compiler and utilities. These techniques are generic and should help boost performance of most applications. You may:

- Provide a RAM disk for the compiler and utilities to use for temporary files,
- Use a disk cache to store the most recently accessed disk files.

### Using a RAM Disk

If your computer has sufficient extended or expanded memory available, you should consider using a RAM disk. A RAM disk is a memory-based disk emulator. Because the contents of a RAM disk are stored in RAM, access is very fast.

If you are using a RAM disk, you can set the value of the **TMP** environment variables to the drive name of the RAM disk. This speeds up the execution of the many of the tools and utilities because they can use the RAM disk for temporary files.

A number of RAM disk software packages are available. **RAMDRIVE.SYS** and **VDISK.SYS** are the names of the RAM disk programs that are most commonly shipped with DOS. Refer to your DOS manual to learn how to install these programs.

## Using a Disk Cache

A disk cache utilizes a large memory pool to temporarily store information read from disk. When the computer accesses the disk, it first checks the cache to see if the desired information is already in the cache. If it is, the information is read from the cache memory instead of from the disk. This is significantly faster than waiting for the disk drive to read the information.

Typically, software development involves an edit-compile-edit-compile... cycle. In these situations, a disk cache improves the performance of your editor, assembler, compiler, and linker. The editor, the compiler, source file, and object file can all be held in the cache, and disk accesses are kept to a minimum.

Version 5.0 and Version 6.0 of MS-DOS both come with a disk-caching utility called `SMARTDRV.SYS`. Refer to your DOS manual to learn how to install and use this program.

## Chapter 3. 8051/251 Product Line

Keil Software provides the premier 8051 and 251 development tools in the industry. To help you become familiar with how we distribute our tools, we would like to introduce the concept of a tool kit.

A tool kit is comprised of several application programs that you use to create your 8051 application. You may use an assembler to assemble your 8051 assembly program, you may use a compiler to compile your C source code into an object file, and you may use a linker to create an absolute object module suitable for your in-circuit emulator.

While it makes little sense to have a compiler without a linker, it also makes little sense to have a linker without a compiler or assembler. Therefore, our tools are packaged into various kits.

Our 8051 kits are described below in the “8051 Development Tool Kits” section. Our 251 kits are described in the “251 Development Tool Kits” section on page 17.

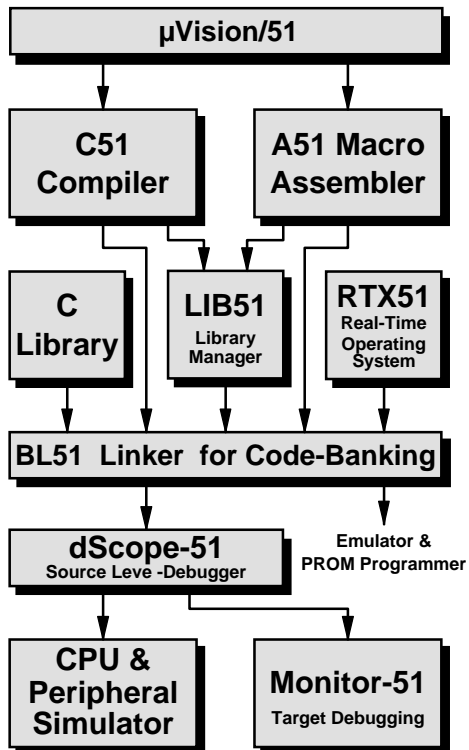
### 8051 Development Tool Kits

When you use the Keil Software tools, the 8051 project development cycle is roughly the same as for any software development project.

1. Create source files in C or assembly.
2. Compile or assemble source files.
3. Correct errors in source files.
4. Link object files from compiler and assembler.
5. Test linked application.

### Tool Kit Overview

The development cycle described above may be best illustrated by a block diagram (shown on the following page) of the complete 8051 tool set.



As shown in this figure, files are created by the  $\mu$ Vision/51 IDE and then passed to the C51 compiler or A51 assembler. The compiler and assembler process source files and create relocatable object files.

Object files created by the compiler and assembler may be used by the LIB51 library manager to create a library. A library is a specially formatted, ordered program collection of object modules that the linker can process. When the linker processes a library, only the object modules in the library that are necessary for program creation are used.

Object files created by the compiler and assembler and library files created by the library manager are processed by the linker to create an

absolute object module. An absolute object file or module is an object file with no relocatable code. All the code in an absolute object file resides at fixed locations.

The absolute object file created by the linker may be used to program EPROM or other memory devices. The absolute object module may also be used with the dScope-51 debugger/simulator or with an in-circuit emulator.

The dScope-51 source level debugger/simulator is ideally suited for fast, reliable high-level-language program debugging. The debugger contains a high-speed simulator and a target debugger that let you simulate an entire 8051 system including on-chip peripherals. By loading specific I/O drivers, you can simulate the attributes and peripherals of a variety of 8051 derivatives. In conjunction with Monitor-51, the debugger is even able to do source-level debugging on your target hardware.

The RTX-51 real-time operating system is a multitasking kernel for the 8051 family. The RTX-51 real-time kernel simplifies the system design, programming, and debugging of complex applications where fast reaction to time critical events is essential. The kernel is fully integrated into the C51 compiler and is easy to use. Task description tables and operating system

consistency are automatically controlled by the BL51 code banking linker/locator.

## Tool Kit Introduction

The preceding diagram shows the full extent of the Keil Software 8051 development tools. The tools listed in this diagram comprise the professional developer's kit described on the following pages. In addition to the professional kit, Keil Software provides a number of other tool kits for the 8051 developer. To best illustrate what is included in each tool kit, we describe the kits in decreasing order of capability. The most capable kit, the professional developer's kit is described first.

**3**

### PK51—C51 Professional Developer's Kit

The PK51 C51 professional developer's kit includes everything the professional 8051 developer needs to create sophisticated embedded applications. This tool kit includes the following components:

- C51 Optimizing C Compiler,
- A51 Macro Assembler,
- BL51 Code Banking Linker/Locator,
- OC51 Banked Object File Converter,
- OH51 Object-Hex Converter,
- LIB51 Library Manager,
- dScope-51 Simulator/Debugger,
- tScope-51 Target Debugger,
- Monitor-51 ROM Monitor and Terminal Program,
- Integrated Development Environment,
- RTX-51 Tiny Real-Time Operating System.

In addition, the professional developer's kit includes the following tools for Windows users:

- dScope-51 Simulator/Debugger for Windows,

- $\mu$ Vision/51 Integrated Development Environment for Windows.

The professional developer's kit can be configured for all 8051 derivatives. The tools included in this kit run under DOS on any 100% IBM PC 386 or higher compatible computer.

### **DK51—C51 Developer's Kit**

The DK51 C51 developer's kit is designed for users who need a complete DOS-based development system for the 8051. This kit lets you create sophisticated embedded applications using a DOS-based development platform. This tool kit includes the following components:

- C51 Optimizing C Compiler,
- A51 Macro Assembler,
- BL51 Code Banking Linker/Locator,
- OC51 Banked Object File Converter,
- OH51 Object-Hex Converter,
- LIB51 Library Manager,
- dScope-51 Simulator/Debugger,
- tScope-51 Target Debugger,
- Monitor-51 ROM Monitor and Terminal Program,
- Integrated Development Environment.

The developer's kit can be configured for all 8051 derivatives. The tools included in this kit run under DOS on any 100% compatible IBM PC 386 or higher computer.

### **CA51—C51 Compiler Kit**

The CA51 C51 compiler kit is the best choice for developers who need a C compiler but not a debugging system. This kit lets you create 8051 C applications for your target hardware. The compiler kit can be configured for all 8051 derivatives. The tools included in this kit run under DOS on any 100% compatible IBM PC 386 or higher computer.

## A51—A51 Macro Assembler Kit

The A51 assembler kit includes our 8051 assembler and all the utilities you need to begin creating 8051 application. The assembler kit is easily configured for all 8051 derivatives. The tools included in this kit run under DOS on any 100% compatible IBM PC 386 or higher computer.

## DS51—dScope-51 Simulator Kit

The DS51 simulator kit provides a debugger/simulator for use with the A51 assembler kit and the CA51 compiler kit. With this kit, you can quickly locate problems in your 8051 application because the simulator lets you step through your code one instruction at a time. You can easily view program variables, SFRs, and memory locations. This tool kit includes the following components:

- dScope-51 Simulator/Debugger,
- tScope-51 Target Debugger,
- Monitor-51 ROM Monitor and Terminal Program.

The simulator kit comes with drivers for most popular 8051 derivatives. The tools included in this kit run under DOS on any 100% compatible IBM PC 386 or higher computer.

## FR51—RTX-51 Full Real-Time Kernel

The RTX-51 Full kernel is a real-time operating system for the 8051 microcontroller. RTX-51 Full provides a superset of the features found in RTX-51 Tiny and also includes BITBUS and CAN communication protocol interface libraries. Refer to “Chapter 8. Real-Time Kernels” on page 111 for more information about RTX-51 Tiny.

# 251 Development Tool Kits

Our 251 development tool set is very similar in function to our 8051 tools set. Where applicable, we have kept the names of the kits the same. The development process for a 251 application is much the same as it is for an 8051 application. The differences are the names of the tools used to generate 251

code. Following are descriptions of the 251 development tool kits that we provide.

### **DK251—C251 Developer's Kit for Windows**

The DK251 C251 developer's kit is designed for users who need a complete development system for the 251. This kit lets you create sophisticated embedded applications using a Windows-based development platform. This tool kit includes the following components:

- C251 Optimizing C Compiler,
- A251 Macro Assembler,
- L251 Linker/Locator,
- OH251 Object-Hex Converter,
- LIB251 Library Manager,
- dScope-251 Simulator/Debugger for Windows,
- Monitor-251 ROM Monitor,
- $\mu$ Vision/251 Integrated Development Environment for Windows.

The developer's kit can be configured for all modes of the 251. The tools included in this kit run under Windows on any 100% IBM PC 386 or higher compatible computer.

### **CA251—C251 Compiler Kit for Windows**

The CA251 compiler kit is the best choice for developers who need a C compiler but not a debugging system. This kit lets you create 251 C and assembly applications for your target hardware. This tool kit includes the following components:

- C251 Optimizing C Compiler,
- A251 Macro Assembler,
- L251 Linker/Locator,
- OH251 Object-Hex Converter,
- LIB251 Library Manager,



- $\mu$ Vision/251 Integrated Development Environment for Windows.

The compiler kit can be configured for every mode of the 251. The tools included in this kit run under Windows on any 100% IBM PC 386 or higher compatible computer.

### **A251—A251 Macro Assembler Kit for Windows**

The A251 assembler kit includes our 251 assembler and all the utilities you need to begin creating 251 application. This tool kit includes the following components:

- A251 Macro Assembler,
- L251 Linker/Locator,
- OH251 Object-Hex Converter,
- LIB251 Library Manager,
- $\mu$ Vision/251 Integrated Development Environment for Windows.

The tools included in this kit run under Windows on any 100% IBM PC 386 or higher compatible computer.

### **DS251—dScope-251 Simulator Kit for Windows**

The DS251 simulator kit provides a Windows-based debugger/simulator for the A251 assembler kit and the CA251 compiler kit. With this kit, you can quickly locate problems in your 251 application because the simulator lets you step through your code one instruction at a time. You can easily view program variables, SFRs, and memory locations. This tool kit includes the following components:

- dScope-251 Simulator/Debugger for Windows,
- Monitor-251 ROM Monitor.

The simulator kit comes with drivers for most popular 8051 and 251 derivatives. The tools included in this kit run under Windows on any 100% IBM PC 386 or higher compatible computer.

## FR251—RTX-251 Full Real-Time Kernel

The RTX-251 Full kernel is a real-time operating system for the 251 microcontroller. RTX-251 Full provides a superset of the features found in RTX-251 Tiny and is comparable to the RTX-51 kernel for the 8051.

## Subscription Kits

### 3

To best support users who develop 8051 and 251 applications, we have added the 8051/251 subscription kits to our product line. The subscription kits provide the 8051 and 251 development tools as well as one year of free software upgrades.

### SDK251—8051/251 Developer's Kit Subscription

The SDK251 subscription includes all the components of the DK51 developer's kit and the DK251 developer's kit. This kit provides a complete solution for 8051 developers who plan to use the 251. The following components are included in this kit.

- C51 Optimizing C Compiler,
- C251 Optimizing C Compiler,
- A51 Macro Assembler,
- A251 Macro Assembler,
- BL51 Code Banking Linker/Locator,
- L251 Linker/Locator,
- OC51 Banked Object File Converter,
- OH51 Object-Hex Converter,
- OH251 Object-Hex Converter,
- LIB51 Library Manager,
- LIB251 Library Manager,
- dScope-51 Simulator/Debugger for DOS,
- tScope-51 Target Debugger for DOS,
- Monitor-51 ROM Monitor and Terminal Program,

- dScope-251 Simulator/Debugger for Windows,
- Monitor-251 ROM Monitor,
- $\mu$ Vision/251 Integrated Development Environment for Windows.

### **SCA251—8051/251 Compiler Kit Subscription**

The SCA251 subscription includes all the components of the CA51 compiler kit and the CA251 compiler kit. This kit is the best choice for 8051 developers who plan to use the 251 but who don't need a debugging solution. The following components are included in this kit.

- C51 Optimizing C Compiler,
- C251 Optimizing C Compiler,
- A51 Macro Assembler,
- A251 Macro Assembler,
- BL51 Code Banking Linker/Locator,
- L251 Linker/Locator,
- OC51 Banked Object File Converter,
- OH51 Object-Hex Converter,
- OH251 Object-Hex Converter,
- LIB51 Library Manager,
- LIB251 Library Manager,
- $\mu$ Vision/251 Integrated Development Environment for Windows.

## Tool Kit Comparison Chart

The following table provides a check list of the features found in each of our development kits. Part numbers are listed across the top and features are listed down the side. Use this cross reference to select the kit that best suits your needs.

Support	PK51	DK51	A51	DK251	CA251	A251	SDK251	SCA251
8051	✓	✓	✓				✓	✓
251				✓	✓	✓	✓	✓
Assembler	✓	✓	✓	✓	✓	✓	✓	✓
Compiler	✓	✓		✓	✓		✓	✓
Simulator	✓	✓		✓			✓	
IDE	✓	✓	✓	✓	✓	✓	✓	✓
RTX	✓							
Windows	✓			✓	✓	✓	✓	✓
DOS	✓	✓	✓				✓	✓

# Chapter 4. 8051 Development Tools

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

## 8051 Microcontroller Family

The 8051 has been available since the early 1980's. With a wide variety of outstanding features and peripherals, the 8051 CPU core is destined to see service well into the next century. More than 200 different 8051 derivatives are available today from a variety of chip vendors. More than half of all embedded projects with a CPU use members of the 8051 microcontroller family. As an embedded processor, the 8051 has no equal.

A typical 8051 family member contains the 8051 CPU core, data memory, code memory, and some versatile peripheral functions. A flexible memory interface lets you expand the capabilities of the 8051 using standard peripherals and memory devices.

## 8051 Development Tools

Keil Software provides the following development tools for the 8051:

- C51 Optimizing C Compiler (see page 24),
- A51 Macro Assembler (see page 41),
- BL51 Code Banking Linker/Locator (see page 43),
- OC51 Banked Object File Converter (see page 47),
- OH51 Object-Hex Converter (see page 47),
- LIB51 Library Manager (see page 47)
- dScope-51 for Windows (see page 47),
- $\mu$ Vision/51 for Windows (see page 48).

For information on the products which include these tools, refer to “Chapter 3. 8051/251 Product Line” on page 13.

---

**NOTE**

*All of our 8051 tools utilize the Intel OMF51 object module format. The development environment can be expanded with all Intel compatible tools such as Intel PL/M-51 or iDCX-51 and with emulators from a wide range of manufactures.*

---

## C51 Optimizing C Cross Compiler

# 4

The C programming language is a general-purpose programming language that provides code efficiency, elements of structured programming, and a rich set of operators. C is not a *big* language and is not designed for any one particular area of application. Its generality, combined with its absence of restrictions, make C a convenient and effective programming solution for a wide variety of software tasks. Many applications can be solved more easily and efficiently with C than with other more specialized languages.

The Keil Software C51 optimizing cross compiler for the MS-DOS operating system is a complete implementation of the ANSI (American National Standards Institute) standard for the C language. The C51 compiler generates code for the 8051 microprocessor but is not a universal C compiler adapted for the 8051 target. It is a ground-up implementation dedicated to generating extremely fast and compact code for the 8051 microprocessor.

For most 8051 applications, the C51 compiler gives software developers the flexibility of programming in C while matching the code efficiency and speed of assembly language.

Using a high-level language like C has many advantages over assembly language programming. For example:

- Knowledge of the processor instruction set is not required. A rudimentary knowledge of the 8051's memory architecture is desirable but not necessary.
- Register allocation and addressing mode details are managed by the compiler.
- The ability to combine variable selection with specific operations improves program readability.
- Keywords and operational functions that more nearly resemble the human thought process can be used.

- Program development and debugging times are dramatically reduced when compared to assembly language programming.
- The library files that are supplied provide many standard routines (such as formatted output, data conversions, and floating-point arithmetic) that may be incorporated into your application.
- Existing routine can be reused in new programs by utilizing the modular programming techniques available with C.
- The C language is very portable and very popular. C compilers are available for almost all target systems. Existing software investments can be quickly and easily converted from or adapted to other processors or environments.

## C51 Language Extensions

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

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

The following sections briefly describe these extensions.

## Data Types

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

Data Type	Bits	Bytes	Value Range
<b>bit</b> †	1		0 to 1
<b>signed char</b>	8	1	-128 to +127
<b>unsigned char</b>	8	1	0 to 255
<b>enum</b>	16	2	-32768 to +32767
<b>signed short</b>	16	2	-32768 to +32767
<b>unsigned short</b>	16	2	0 to 65535
<b>signed int</b>	16	2	-32768 to +32767
<b>unsigned int</b>	16	2	0 to 65535
<b>signed long</b>	32	4	-2147483648 to 2147483647
<b>unsigned long</b>	32	4	0 to 4294967295
<b>float</b>	32	4	$\pm 1.175494\text{E}-38$ to $\pm 3.402823\text{E}+38$
<b>sbit</b> †	1		0 to 1
<b>sfr</b> †	8	1	0 to 255
<b>sfr16</b> †	16	2	0 to 65535

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

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

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

## Memory Types

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



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

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

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

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

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

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

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

## Memory Models

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

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

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

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

---

### NOTE

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

---

## Pointers

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

### Generic Pointers

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

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

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

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

### Memory Specific Pointers

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

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

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

## Comparison: Memory Specific & Generic Pointers

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

Description	Idata Pointer	Xdata Pointer	Generic Pointer
Sample Program	<code>char idata *ip; char val; val = *ip;</code>	<code>char xdata *xp; char val; val = *xp;</code>	<code>char *p; char val; val = *p;</code>
8051 Program Code Generated	<code>MOV R0,ip MOV val,@R0</code>	<code>MOV DPL,xp +1 MOV DPH,xp MOV A,@DPTR MOV val,A</code>	<code>MOV R1,p + 2 MOV R2,p + 1 MOV R3,p CALL CLDPTR</code>
Pointer Size	1 byte data	2 bytes data	3 bytes data
Code Size	4 bytes code	9 bytes code	11 bytes code + Lib.
Execution Time	4 cycles	7 cycles	13 cycles

# 4

## Reentrant Functions

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

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

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

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

**NOTE**

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

## Interrupt Functions

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

```

unsigned int interruptcnt;
unsigned char second;

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

```

## Parameter Passing

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

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

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

## Function Return Values

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

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

# 4

## Register Optimizing

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

## Real-Time Operating System Support

The C51 compiler integrates well with both the RTX-51 Full and RTX-51 Tiny multitasking real-time operating systems. The task description tables are generated and controlled during the link process. For more information about the RTX real-time operating systems, refer to “Chapter 8. Real-Time Kernels” on page 111.

## Interfacing to Assembly

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

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

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

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

```
?PR?_asmfunc1?ASM1      SEGMENT CODE
PUBLIC                  _asmfunc1
                        RSEG ?PR?_asmfunc1?ASM1
                        USING 0

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

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

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

## Interfacing to PL/M-51

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

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

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



## Code Optimizations

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

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

### General Optimizations

- **Constant Folding:** Several constant values occurring in an expression or address calculation are combined as a single constant.
- **Jump Optimizing:** Jumps are inverted or extended to the final target address when the program efficiency is thereby increased.
- **Dead Code Elimination:** Code which cannot be reached (dead code) is removed from the program.
- **Register Variables:** Automatic variables and function arguments are located in registers whenever possible. No data memory space is reserved for these variables.
- **Parameter Passing Via Registers:** A maximum of three function arguments can be passed in registers.
- **Global Common Subexpression Elimination:** Identical subexpressions or address calculations that occur multiple times in a function are recognized and calculated only once whenever possible.

## 8051-Specific Optimizations

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

# 4

## Options for Code Generation

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

## Global Register Optimization

The C51 compiler provides support for application wide register optimization which is also known as application register coloring. The following sample program compares the code generated by C51 version 5.0 using application register coloring to the code generated by C51 version 3.4 without application register coloring.

With the application wide register optimization, the C compiler *knows* the registers used by external functions. Registers which 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. In the following

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

With Global Register Optimization	Without Global Register Optimization
<pre>main () {     unsigned char i;     unsigned char a;     while (1) {         i = input (); </pre>	
<pre>?C0001:     LCALL    input ;- 'i' assigned to 'R6' -     MOV     R6,AR7</pre>	<pre>?C0001:     LCALL    input     MOV     DPTR,#i     MOV     A,R7     MOV     @DPTR,A</pre>
<pre>        do {             a = input (); </pre>	
<pre>?C0005:     LCALL    input ;- 'a' assigned to 'R7' -     MOV     R5,AR7</pre>	<pre>?C0005:     LCALL    input     MOV     DPTR,#a     MOV     A,R7     MOVX    @DPTR,A</pre>
<pre>            output (a); </pre>	
<pre>    LCALL    _output</pre>	<pre>    LCALL    _output</pre>
<pre>        } while (--i); </pre>	
<pre>DJNZ     R6,?C0005</pre>	<pre>MOV     DPTR,#i MOVX    A,@DPTR DEC     A MOVX    @DPTR,A JNZ     ?C0005</pre>
<pre>    } </pre>	
<pre>SJMP     ?C0001</pre>	<pre>SJMP     ?C0001</pre>
<pre>    } </pre>	
<pre>RET</pre>	<pre>RET</pre>
Code Size: 18 Bytes	Code Size: 30 Bytes

4

## Debugging

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

compatible with the Intel OMF51 object module format and if it can accept Keil object modules.

## Library Routines

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

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

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

## Intrinsic Library Routines

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

Intrinsic Function	Description
<code>_crol_</code>	Rotate character left.
<code>_cror_</code>	Rotate character right.
<code>_irol_</code>	Rotate integer left.
<code>_iror_</code>	Rotate integer right.
<code>_lrol_</code>	Rotate long integer left.
<code>_lror_</code>	Rotate long integer right.
<code>_nop_</code>	No operation (8051 NOP instruction).
<code>_testbit_</code>	Test and clear bit (8051 JBC instruction).

## Listing File Example

The C51 compiler produces a listing file that contains source code, directive information, an assembly listing, and a symbol table.

```

C51 COMPILER V5.02, SAMPLE 07/01/95 08:00:00 PAGE 1
DOS C51 COMPILER V5.02, COMPILATION OF MODULE SAMPLE
OBJECT MODULE PLACED IN SAMPLE.OBJ
COMPILER INVOKED BY: C:\C51\BIN\C51.EXE SAMPLE.C CODE

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

ASSEMBLY LISTING OF GENERATED OBJECT CODE

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

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

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

```

The C51 compiler produces a listing file with page numbers as well as time and date of the compilation. Remarks about the compiler invocation and object file output are displayed in this listing.

The listing includes a line number for each statement and a nesting level for each block enclosed within curly braces ('{' and '}').

Error messages and warning messages are included in the listing file.

The **CODE** compiler option includes an assembly code listing in the listing file. Source line numbers are embedded within the generated code.

A memory overview provides information about the 8051 memory areas that are used.

The total number of errors and warnings is stated at the end of the listing file.

# A51 Macro Assembler

The A51 assembler is a macro assembler for the 8051 microcontroller family. It translates symbolic assembly language mnemonics into relocatable object code where the utmost speed, small code size, and hardware control are critical. The macro facility speeds development and conserves maintenance time since common sequences need only be developed once. The A51 assembler supports symbolic access to all features of the 8051 architecture and is configurable for the numerous 8051 derivatives.

## Functional Overview

The A51 assembler translates an assembler source file into a relocatable object module. If the **DEBUG** control is used, the object file contains full symbolic information for debugging with dScope or an in-circuit emulator. In addition to the object file, the A51 assembler generates a list file which may optionally include symbol table and cross reference information. The A51 assembler is fully compatible with Intel ASM-51 source modules.

**4**

## Configuration

The A51 assembler supports all members of the 8051 family. The special function register (SFR) set of the 8051 is predefined. However, the **NOMOD51** control lets you override these definitions with processor-specific include files. The A51 assembler is shipped with include files for the 8051, 8051Fx, 8051GB, 8052, 80152, 80451, 80452, 80515, 80C517, 80C515A, 80C517A, 8x552, 8xC592, 8xCL781, 8xCL410 and 80C320 microcontrollers. You can easily create include files for other 8051 family members.

## Listing File Example

The following example shows a listing file generated by the A51 assembler during assembly. The listing file contains source code, machine code generated, directive information, and a symbol table.

```

A51 MACRO ASSEMBLER Test Program 07/01/95 08:00:00 PAGE 1
DOS MACRO ASSEMBLER A51 V5.02
OBJECT MODULE PLACED IN SAMPLE.OBJ
ASSEMBLER INVOKED BY: C:\C51\BIN\A51.EXE SAMPLE.A51 XREF

LOC OBJ LINE SOURCE
1 $TITLE ('Test Program')
2 NAME SAMPLE
3
4 EXTRN CODE (PUT_CRLF, PUTSTRING, InitSerial)
5 PUBLIC TXTBIT
6
7 PROG SEGMENT CODE
8 CONST SEGMENT CODE
9 BITVAR SEGMENT BIT
10
---- 11 CSEG AT 0
12
0000 020000 F 13 Reset: JMP Start
14
---- 15 RSEG PROG
16 ; *****
0000 120000 F 17 Start: CALL InitSerial ;Init Serial Interface
18
19 ; This is the main program. It is an endless
20 ; loop which displays a text on the console.
0003 C200 F 21 CLR TXTBIT ; read from CODE
0005 900000 F 22 Repeat: MOV DPTR,#TXT
0008 120000 F 23 CALL PUTSTRING
000B 120000 F 24 CALL PUT_CRLF
000E 80F5 25 SJMP Repeat
26 ;
---- 27 RSEG CONST
0000 5445354 28 TXT: DB 'TEST PROGRAM',00H
0004 2050524F
0008 4752414D
000C 00
29
30
31
---- 32 RSEG BITVAR ; TXTBIT=0 read from CODE
0000 33 TXTBIT: DBIT 1 ; TXTBIT=1 read from XDATA
34
35 END

XREF SYMBOL TABLE LISTING
-----
N A M E T Y P E V A L U E A T T R I B U T E S / R E F E R E N C E S

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

REGISTER BANK(S) USED: 0
ASSEMBLY COMPLETE. 0 WARNING(S), 0 ERROR(S)

```

The A51 assembler produces a listing file with page numbers as well as the time and date of the assembly. Remarks about the assembler invocation and the object file output are displayed in this listing.

Typical programs start with EXTERN, PUBLIC, and SEGMENT directives.

The listing file includes a line number for each source line.

If a source line generates code, the HEX values are displayed at the beginning of the line.

Error messages and warning messages are included in the listing file. The position of each error is clearly marked.

The XREF assembler option produces a cross reference list. The cross reference report shows all symbols and the line numbers in which they are used. The line number where the symbol is defined is marked with a pound symbol ('#').

The register banks used, and the total number of warnings and errors is stated at the end of the listing file.



## BL51 Code Banking Linker/Locator

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

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

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

**4**

## Data Address Management

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

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

## Code Banking

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

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

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

## Common Area

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

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

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

## Executing Functions in Other Banks

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

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

The bank switching process requires approximately 50 CPU cycles and consumes an additional 2 bytes of stack space. You can dramatically improve system performance by grouping interdependent functions in the same bank.

Functions which are frequently invoked from multiple banks should be located in the common area.

## Listing File Example

The following example shows a map file created by the BL51 code banking linker/locator:

```

BL51 BANKED LINKER/LOCATER V3.52          07/01/95  08:00:00  PAGE 1
MS-DOS BL51 BANKED LINKER/LOCATER V3.52, INVOKED BY:
C:\C51\BIN\BL51.EXE SAMPLE.OBJ

MEMORY MODEL: SMALL

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

LINK MAP OF MODULE:  SAMPLE (SAMPLE)

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

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

OVERLAY MAP OF MODULE:  SAMPLE (SAMPLE)

SEGMENT          DATA_GROUP
+--> CALLED SEGMENT      START      LENGTH
-----
?C_C51STARTUP
+--> ?PR?MAIN?SAMPLE
?PR?MAIN?SAMPLE          0009H      0001H
+--> ?PR?GETCHAR?GETCHAR
+--> ?PR?_TOUPPER?TOUPPER
+--> ?PR?PUTCHAR?PUTCHAR
?PR?GETCHAR?GETCHAR
+--> ?PR?_GETKEY?_GETKEY
+--> ?PR?PUTCHAR?PUTCHAR

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

```

The BL51 code banking linker/locator produces a map file with the time and date of the link/locate run.

The invocation line and the selected memory module are listed.

The link-map contains a table of the memory usage of the physical 8051 memory area.

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

Error messages and warnings are listed at the end of the map file. These messages

indicate possible problems during the link/locate run.

## OC51 Banked Object File Converter

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

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

## OH51 Object-Hex Converter

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

## LIB51 Library Manager

The LIB51 library manager lets you create and maintain library files. A library file is a formatted collection of one or more object files. Library files provide a convenient method of combining and referencing a large number of object files. Libraries can be effectively used by the BL51 code banking linker/locator.

The LIB51 library manager lets you create a library file, add object modules to a library file, remove object modules from a library file, and list the contents of a library file. The LIB51 library manager may be controlled interactively or from the command line.

## dScope-51 for Windows

dScope-51 is a source level debugger and simulator for programs created with the Keil C51 compiler and A51 assembler and the Intel PL/M-51 compiler and ASM-51 assembler. dScope-51 is a software-only product that lets you simulate

the features of an 8051 without actually having target hardware. You may use dScope-51 to test and debug your embedded application before actual 8051 hardware is ready. dScope-51 simulates a wide variety of 8051 peripherals including the internal serial port, external I/O, and timers.

---

**NOTE**

*dScope-51 and dScope-251 are essentially the same product. The only differences are the support for either 8051 or 251 development tools. dScope is used throughout this book to refer to either debugger.*

---

Refer to “dScope Simulator/Debugger Overview” on page 70 for examples that show how to use dScope-51.

## 4

### μVision/51 for Windows

μVision/51 is an integrated software development platform that includes a full-function editor, project manager, make facility, and environment control for the Keil 8051 tools. When you use μVision/51, you no longer have to learn the command-line syntax of any of the tools. μVision/51 speeds your embedded application development by providing the following:

- Standard Windows user interface,
- Dialog boxes for all environment and development tool settings,
- Multiple file editing capability,
- Full-function editor with user-definable key sequences,
- Application manager for adding external programs into the pull-down menu,
- Project manager for creating and maintaining projects,
- Integrated make facility for building target programs from your projects,
- On-line help system.

---

**NOTE**

*μVision/51 and μVision/251 are essentially the same product. The only differences are the support for either 8051 or 251 development tools. μVision is used throughout this book to refer to either IDE.*

---

Refer to “μVision IDE Overview” on page 64 for examples that show how to use μVision/51.



**4**



## Chapter 5. 251 Development Tools

This chapter discusses the features and advantages of the 8051 and MCS<sup>®</sup> 251 microprocessor family and the development tools available from Keil Software. We have designed our development tools to help you quickly and successfully complete your job. For this reason, our tools are easy to use and are guaranteed to help you achieve your design goals.

### MCS<sup>®</sup> 251 Microcontroller Family

In response to demands for more power and capability, Intel has developed the new MCS<sup>®</sup> 251 microcontroller family. The 251 is completely backwards compatible with the 8051. All of your existing 8051 software can run on the new 251 CPU. Intel's first derivative, the 80C251SB, is a direct pin-for-pin replacement for the 80C51FX.

By design, the 251 is a powerful 8-bit/16-bit CPU. At an equivalent clock speed, existing 8051 code executes up to 5 times faster on the 251. C applications that are recompiled with Keil C251 can execute up to 15 times faster. Your existing 8051 software can achieve dramatic performance increases by using the features found in the 251 instruction set. The following are a few highlights of the 251:

- 8-bit, 16-bit, and 32-bit instructions,
- 8-bit, 16-bit, and 32-bit registers,
- 16 Mbyte linear address space,
- Direct support for 16-bit and 32-bit pointers,
- 16-bit stack & stack addressing,
- Direct addressing mode for 64KB.

In addition to these features, the 251 offers configuration options for binary mode (8051 compatible instruction set) or source mode (251 instruction set), page or non-page mode, and wait state generation.

## 251 Development Tools

Keil Software provides the following development tools for the 251:

- C251 Optimizing C Compiler (see below),
- A251 Macro Assembler (see page 55),
- L251 Linker/Locator (see page 59),
- OH251 Object HEX Converter (see page 60),
- LIB251 Library Manager (see page 60).
- dScope-251 for Windows (see page 60),
- $\mu$ Vision/251 for Windows (see page 61).

For information on the products which include these tools, refer to “Chapter 3. 8051/251 Product Line” on page 13.

---

### **NOTE**

*All of our 251 tools utilize the Intel OMF251 object module format. Additionally, the L251 linker/locator can combine OMF251 and OMF51 object modules.*

---

# 5

## C251 Optimizing C Cross Compiler

The Keil C251 Compiler is a dedicated ANSI C compiler designed explicitly for the MCS<sup>®</sup> 251 microcontroller family. The C251 compiler is an extended ANSI C compiler which allows full access to all resources in a 251 microcontroller system. You may re-compile your existing C51 code with the C251 compiler.

## Data Types

The C251 compiler supports the following data types.

Data Type	Bits	Bytes	Value Range
<b>bit</b>	1		0 or 1
<b>signed char</b>	8	1	-128 to +127
<b>unsigned char</b>	8	1	0 to 255
<b>signed int</b>	16	2	-32768 to +32767
<b>unsigned int</b>	16	2	0 to 65535
<b>signed long</b>	32	4	-2147483648 to +2147483647
<b>unsigned long</b>	32	4	0 to 4294967295
<b>float</b>	32	4	$\pm 1.175494E-38$ to $\pm 3.402823E+38$
<b>double</b>	64	8	$\pm 1.7E-308$ to $\pm 1.7E+308$
<b>pointer</b>		1, 2, 3, or 4	Object address
<b>sbit</b>	1		0 or 1
<b>sfr</b>	8	1	0 to 255
<b>sfr16</b>	16	2	0 to 65535

## Memory Selector

The C251 compiler provides full support for the 251 architecture and can access all system components. Each variable can be explicitly located anywhere in the 251 address space. The linear 16 Mbyte address space can be accessed with many addressing modes. In addition, all addressing modes of the 8051 are fully supported by the 251.

Selector	251 Address Space
<b>near</b>	64 Kbyte direct and indirect memory addressing.
<b>far</b>	16 Mbyte indirect memory addressing; object size < 64 Kbytes.
<b>huge</b>	16 Mbyte indirect memory addressing; any object size.
<b>data</b>	Direct memory addressing for on-chip RAM (128 bytes); fast 8-bit accesses.
<b>bdata</b>	Bit-addressable RAM; mixed bit and byte accesses (16 bytes).
<b>ebdata, ebit</b>	Extended bit-addressable RAM; mixed bit and byte accesses.
<b>idata</b>	Indirect memory addressing for on-chip RAM (256 bytes); access with <b>MOV @Ri</b> .
<b>pdata</b>	Paged <b>XDATA</b> memory (256 bytes); access with <b>MOVX @Ri</b> .
<b>xdata</b>	<b>XDATA</b> memory (64 Kbytes); access with <b>MOVX @DPTR</b> .
<b>code</b>	<b>CODE</b> memory (64 Kbytes); access with <b>MOVC</b> .

## Memory Models

The memory model determines the default memory selector used for automatic variables and parameter passing areas. With the **HOLD** directive you can specify additional memory selectors for small objects, for example, the following command line:

```
C251 PROG.C HOLD ( 2, 4, 8 )
```

directs the C251 compiler to locate global variables 2 bytes in size or smaller in **data** memory; variables 3 or 4 bytes in size in **near** memory; and variables 5 to 8 bytes in size in **xdata** memory.

The following table lists the memory areas used for each memory model.

Memory Model	Parameters & Automatic Variables	Global Variables	Constant Variables	Pointer Definition	Pointer Size
TINY	data	data	near	near *	2 bytes
SMALL	data	data	near	far *	3 / 4 bytes
COMPACT	pdata	pdata	near	far *	3 / 4bytes
MEDIUM	near	near	far	far *	3 / 4 bytes
LARGE	xdata	xdata	near	far *	3 / 4 bytes

5

## Program Size

The MCS<sup>®</sup> 251 microcontroller family allows program sizes up to 16 Mbytes. The generated 251 code can be optimized by using specific **JMP** and **CALL** instructions. The **ROM** directive lets you choose the combination of **JMP** and **CALL** instructions that is used.

ROM Directive	JMP Instruction	CALL Instruction
SMALL	AJMP	ACALL
COMPACT	AJMP	LCALL
LARGE	LJMP	LCALL
HUGE	LJMP	ECALL

## Register Optimization

Depending on the program context, the C251 compiler allocates up to 24 CPU registers for register variables. Any registers modified during function execution are noted within each module. The linker/locator generates a global, project-wide register file which contains information about the registers altered by external functions. Consequently, the C251 compiler *knows* the registers used by each function in an application. With this information, the C251 compiler can optimize the overall CPU register allocation of those functions.

Registers R0-R7 are used for parameter passing. This technique yields very efficient code that compares favorably to assembly programming. Additional parameters are passed via fixed memory locations or the 251's hardware stack.

## Reentrant Code

The 251 supports stack-based variable addressing. This permits the C251 compiler to support fast reentrant functions. The **#pragma reentrant** and **#pragma noreentrant** preprocessor directives control code generation. Non-reentrant code stores variables in directly addressable memory locations and yields the fastest program execution. Data overlaying considerably reduces the memory requirements in C applications of this type.

**5**

## C Run-Time Library

The run-time libraries provided with the C251 compiler contain over 100 routines, all of which are reentrant. Source code for I/O and memory allocation functions is also included.

## Listing File Example

The C251 compiler produces a listing file that contains source code, directive information, an assembly listing, and a symbol table.

```

C251 COMPILER V1.00, SAMPLE          07/01/95 08:00:00 PAGE 1
DOS C251 COMPILER V1.00, COMPILATION OF MODULE SAMPLE
OBJECT MODULE PLACED IN SAMPLE.OBJ
COMPILER INVOKED BY: C:\C251\BIN\C251.EXE SAMPLE.C CODE

stmt level source

 1      #include <reg251sb.h> /* SFRs for the 251SB CPU */
 2
 3      unsigned long outsqr (
 4          unsigned long num,
 5          unsigned power) {
 6  1
 7  1      unsigned long result;
 8  1
 9  1      for (result = 1; power != 0; power--) {
10  2          result *= num;
11  2      }
12  1
13  1      return (result);
14  1      }
15

ASSEMBLY LISTING OF GENERATED OBJECT CODE

          ; FUNCTION OUTSQR (BEGIN)
0000 A57A1D00 R    MOV    num,DR4
                                ; SOURCE LINE # 3
                                ; SOURCE LINE # 5
                                ; SOURCE LINE # 9
0004 A56D55          XRL    WR10,WR10
0007 A57A5500 R    MOV    result,WR10
000B A57E540001     MOV    WR10,#01H
0010 A57A5500 R    MOV    result+02H,WR10
0014          ?C0001:
0014 E500          R    MOV    A,power+01H
0016 4500          R    ORL    A,power
0018 6019          JZ     ?C0002
                                ; SOURCE LINE # 10
001A A57E1D00 R    MOV    DR4,result
001E A57E0D00 R    MOV    DR0,num
0022 120000 E    LCALL  ?C?LMUL
0025 A57A1D00 R    MOV    result,DR4
                                ; SOURCE LINE # 11
0029 E500          R    MOV    A,power+01H
002B 1500          R    DEC    power+01H
002D 7002          JNZ    ?C0005
002F 1500          R    DEC    power
0031          ?C0005:
0031 80E1          SJMP   ?C0001
0033          ?C0002:
                                ; SOURCE LINE # 13
0033 A57E1D00 R    MOV    DR4,result
                                ; SOURCE LINE # 14
0037          ?C0004:
0037 22          RET
          ; FUNCTION OUTSQR (END)

MODULE INFORMATION:  STATIC OVERLAYABLE
CODE SIZE          =      56  ----
CONSTANT SIZE      =  ----  ----
XDATA SIZE         =  ----  ----
PDATA SIZE         =  ----  ----
DATA SIZE          =  ----  10
IDATA SIZE         =  ----  ----
BIT SIZE           =  ----  ----
EDATA SIZE         =  ----  ----
FDATA SIZE        =  ----  ----
END OF MODULE INFORMATION.

C251 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)

```

The C251 compiler produces a listing file with page numbers as well as time and date of the compilation. Remarks about the compiler invocation and object file output are displayed in this listing.

The listing includes a line number for each statement and a nesting level for each block enclosed within curly braces ('{' and '}').

Error messages and warning messages are included in the listing file.

The **CODE** compiler option includes an assembly code listing in the listing file. Source line numbers are embedded within the generated code.

A memory overview provides information about the 251 memory areas that are used.

The total number of errors

---

and warnings is stated at

the end of the listing file.

## A251 Macro Assembler

The A251 assembler is a macro assembler for the 251 family which supports both the Intel Macro Programming Language (MPL) and Microsoft style macros. The A251 assembler translates symbolic assembly mnemonics into executable machine code. With the A251 assembler, you can easily re-assemble source code written for the Keil A51 assembler or the Intel ASM-51 assembler.

### Functional Overview

The A251 assembler translates 251 assembly source file into relocatable object modules. If the **DEBUG** control is used, object files contain full symbolic information for debugging with dScope or an in-circuit emulator. The A251 assembler also generates a list file which may include symbol table and cross reference listings.

## 5

### Listing File Example

The following example shows a listing file generated by the A251 assembler during assembly. The listing file contains source code, machine code generated, directive information, and a symbol table.

```
A251 MACRO ASSEMBLER  Test Program          07/01/95 08:00:00 PAGE      1
DOS MACRO ASSEMBLER A251 V1.00
OBJECT MODULE PLACED IN SAMPLE.OBJ
ASSEMBLER INVOKED BY: C:\C251BIN\A251.EXE SAMPLE.A51

LOC      OBJ          LINE  SOURCE
-----
                1  $TITLE ('Test Program')
                2  $MODSRC
                3  NAME      SAMPLE
                4
                5  EXTRN   CODE (PUT_CRLF, PUTSTRING, InitSerial)
                6  PUBLIC  TXTBIT
                7
-----
                8  PROG    SEGMENT      CODE
                9  STRINGS SEGMENT      CODE
-----
               10  BITVAR  SEGMENT      BIT
               11
000000        12          CSEG  AT      0
               13
000000 020000  F 14  Reset:  JMP      Start
               15
-----
               16          RSEG  PROG
               17  ; *****
000000 120000  E 18  Start:  CALL  InitSerial ;Init Serial Interface
               19
```

The A251 assembler produces a listing file with page numbers as well as the time and date of the assembly. Remarks about the assembler invocation and the object file output are displayed in this listing.

Typical programs start with **EXTERN**, **PUBLIC**, and **SEGMENT** directives.

The listing file includes a line number for each source line.



```

                20 ; This is the main program. It is an endless
                21 ; loop which displays a text on the console.
000003 C200 F 22 CLR TXTBIT ; read from CODE
000005 900000 F 23 Repeat: MOV DPTR,#TXT
000008 120000 E 24 CALL PUTSTRING
00000B 120000 E 25 CALL PUT_CRLF
00000E 8000 F 26 SJMP Repeat
                27 ;
                28 RSEG STRINGS
000000 54455354 29 TXT: DB 'TEST PROGRAM',00H
000004 2050524F
000008 4752414D
00000C 00

                30
                31
                32
-----
                33 RSEG BITVAR ; TXTBIT=0 read from CODE
0000.0 34 TXTBIT: DBIT 1 ; TXTBIT=1 read from XDATA
                35
                36 END

SYMBOL TABLE LISTING
-----
N A M E T Y P E V A L U E A T T R I B U T E S
BITVAR . . . . . B SEG 000001H REL=UNIT, ALN=BIT
INITSERIAL . . . . C ADDR ----- EXT
PROG . . . . . C SEG 000010H REL=UNIT, ALN=BYTE
PUTSTRING. . . . . C ADDR ----- EXT
PUT_CRLF . . . . . C ADDR ----- EXT
REPEAT . . . . . C ADDR 0005H R SEG=PROG
RESET . . . . . C ADDR 0000H R SEG=?CO?SAMPLE?4
SAMPLE . . . . . -- ---- -----
START . . . . . C ADDR 0000H R SEG=PROG
STRINGS . . . . . C SEG 00000DH REL=UNIT, ALN=BYTE
TXT . . . . . C ADDR 0000H R SEG=STRINGS
TXTBIT . . . . . B ADDR 0000H.0 R SEG=BITVAR

REGISTER BANK(S) USED: 0

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

```

If a source line generates code, the HEX values are displayed at the beginning of the line.

Error messages and warning messages are included in the listing file. The position of each error is clearly marked.

The symbol table listing includes names of data objects, the object type, address, and other attributes.

The register banks used, and the total number of warnings and errors is stated at the end of the listing file.

5

## L251 Code Banking Linker/Locator

The L251 linker/locator is a code banking linker for our 251-based tools. The L251 linker lets you combine object modules created by our 251 tools with object modules created with our 8051 tools. The L251 linker/locator combines one or more object modules into a single executable 251 program. The linker also resolves external and public references, and assigns absolute addresses to relocatable programs segments.

The L251 linker/locator processes object modules created by the Keil C51 compiler, C251 compiler, A51 assembler, and A251 assembler and also processes object modules created by the Intel PL/M-51 compiler and ASM-51 assembler. The linker automatically selects the appropriate run-time library and links only the library modules that are required.

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

## OH251 Object-Hex Converter

The OH251 object-hex converter creates Intel HEX files from OMF251 absolute object modules. Absolute object modules are typically created by the L251 linker/locator. Intel HEX files are ASCII files that contain a hexadecimal representation of your application. They can be easily loaded into a device programmer for writing EPROMS.

## LIB251 Library Manager

### 5

The LIB251 library manager lets you create and maintain library files. A library file is a formatted collection of one or more object files. Library files provide a convenient method of combining and referencing a large number of object files. Libraries can be effectively used by the L251 linker/locator.

The LIB251 library manager lets you create a library file, add object modules to a library file, remove object modules from a library file, and list the contents of a library file. The LIB251 library manager may be controlled interactively or from the command line.

## dScope-251 for Windows

dScope-251 is a source level debugger and simulator for programs created with the Keil C251 compiler and A251 assembler. dScope-251 is a software-only product that lets you simulate the features of the 251 without actually having target hardware. You may use dScope-251 to test and debug your embedded application before actual 251 hardware is ready. dScope-251 simulates all peripherals of the 251 including the 16 Mbyte address space.

---

**NOTE**

*dScope-51 and dScope-251 are essentially the same product. The only differences are the support for either 8051 or 251 development tools. dScope is used throughout this book to refer to either debugger.*

---

Refer to “dScope Simulator/Debugger Overview” on page 70 for examples that show how to use dScope-251.

## **μVision/251 for Windows**

μVision/251 is an integrated software development platform that includes a full-function editor, project manager, make facility, and environment control for the Keil 251 tools. μVision/251 speeds your embedded application development by providing the following:

- Standard Windows user interface,
- Dialog boxes for all environment and development tool settings,
- Multiple file editing capability,
- Full-function editor with user-definable key sequences,
- Application manager for adding external programs into the pull-down menu,
- Project manager for creating and maintaining projects,
- Integrated make facility for building target programs from your projects,
- On-line help system.

---

**NOTE**

*μVision/51 and μVision/251 are essentially the same product. The only differences are the support for either 8051 or 251 development tools. μVision is used throughout this book to refer to either IDE.*

---

Refer to “μVision IDE Overview” on page 64 for examples that show how to use μVision/251.



## Chapter 6. Using the 8051/251 tools

To make it easy for you to evaluate and become familiar with our 8051 and 251 product line, we provide an evaluation diskette with sample programs and limited versions of our tools. The sample programs are also included with our standard product kits.

This chapter introduces the primary user-interface products,  $\mu$ Vision and dScope, and shows you how to use them to compile, link, and run the provided sample programs. The following sections are included in this chapter:

- Starting  $\mu$ Vision and dScope,
- $\mu$ Vision integrated development environment overview,
- dScope simulator/debugger overview,
- Sample programs,
- Building and running the HELLO sample program,
- Building and running the MEASURE sample program,
- Building the BADCODE sample program.

The examples and descriptions in this chapter are illustrated using our Windows-based tools. These are the same tools distributed with our 8051/251 Evaluation Kit. Contact sales/support if you would like a copy of our DOS-based evaluation kit.

---

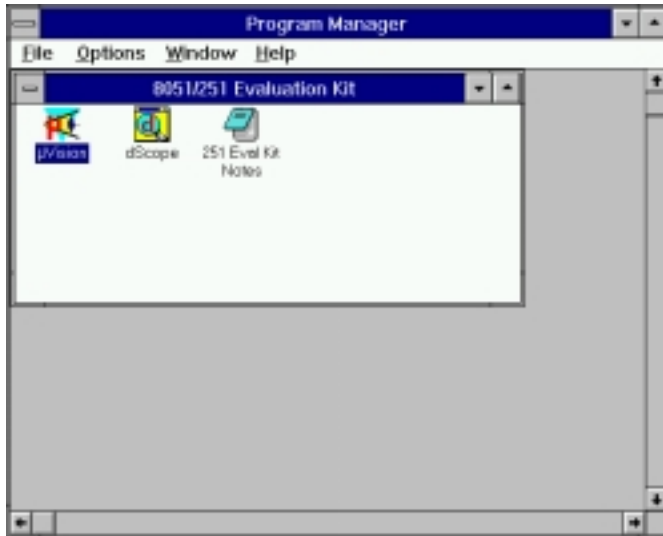
### **NOTE**

*The 8051/251 Evaluation Kit includes evaluation versions of our 8051/251 tools. The evaluation tools are limited in functionality and the code size of the application you can create. Refer to the “Eval Kit 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 “Chapter 3. 8051/251 Product Line” on page 13 for a description of the kits that are available.*

---

## Starting $\mu$ Vision and dScope

Both  $\mu$ Vision for Windows and dScope for Windows are standard Windows applications. You launch them by double-clicking on the appropriate icon in the program group created by the installation program.



## 6

### $\mu$ Vision IDE Overview

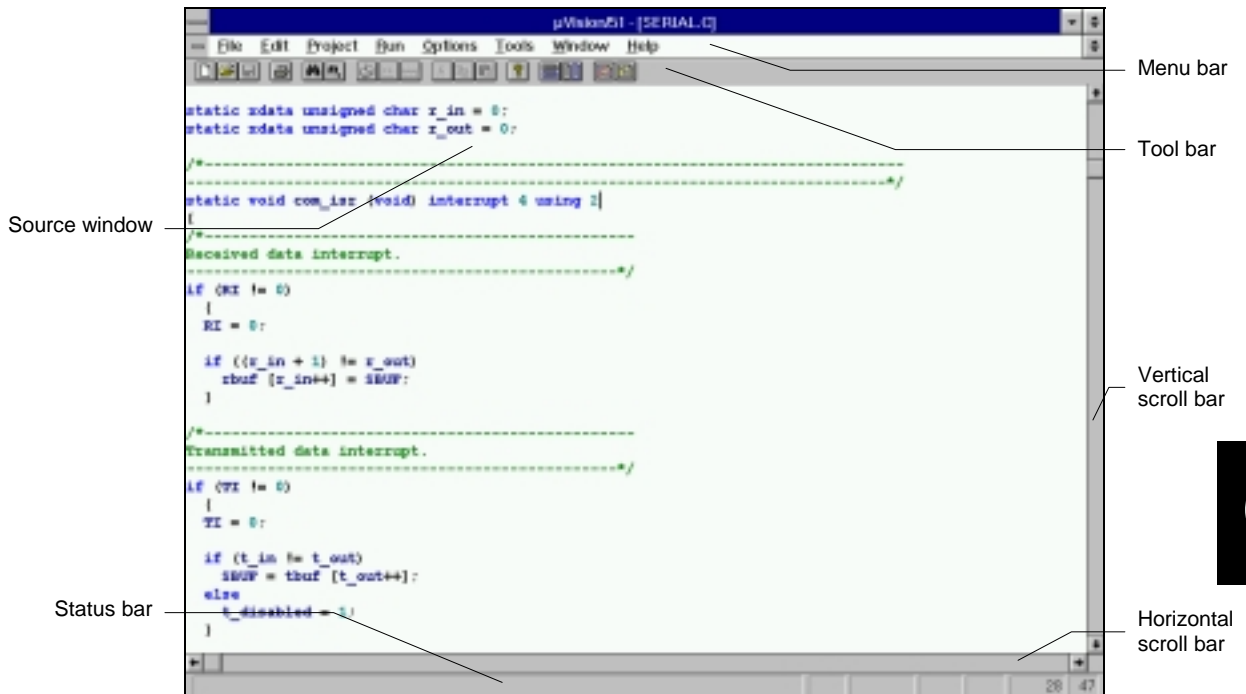
$\mu$ Vision is an integrated software development platform that combines a robust editor, project manager, and make facility.  $\mu$ Vision supports all of the Keil tools for the 8051, 251, and 166.  $\mu$ Vision helps expedite the development process of your embedded applications by providing the following:

- Full-function editor with user-definable key sequences,
- Application manager for linking external program files into the pull-down menu,
- Project manager for creating and maintaining your projects,
- Integrated make facility for assembling, compiling, and linking your embedded applications,
- Dialog boxes for all environment and development tool settings.

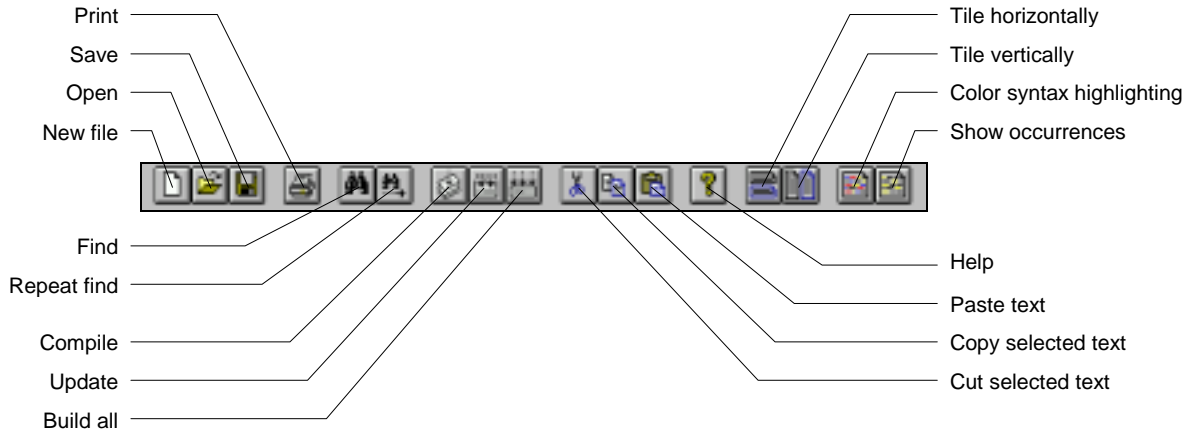
## About the Environment

In  $\mu$ Vision, you may use the keyboard or the mouse to select menu commands, settings, and options for the development tools. You may also use the keyboard to enter program text.

The  $\mu$ Vision screen provides you with a menu bar for command entry, a tool bar where you can rapidly select command buttons, and one or more windows for source files, dialog boxes, and information displays.



You can quickly access many of the features of  $\mu$ Vision using the buttons on the tool bar.



$\mu$ Vision lets you simultaneously open and view multiple source files. While writing part of your C program in one window, you can refer to header file information in another window. You can move and resize source windows using the mouse or keyboard.

6

The screenshot shows the  $\mu$ Vision51 IDE with the following code visible in the windows:

```

SERIAL.C
static xdata unsigned char x_in = 0;
static xdata unsigned char x_out = 0;

/*-----
static void com_isr (void) interrupt 4 using 3
{
/*
Received data interrupt.
if (RI != 0)
{
RI = 0;

if ((x_in + 1) != x_out)
xbuf [x_in++] = SBUF;
}
}
/*-----
Transmitted data interrupt.
if (TI != 0)
{
TI = 0;

if (t_in != t_out)
SBUF = tbuf [t_out++];
else
t_disabled = 1;
}
}
}

TDP.H
MAIN.C
void main (void)
{
/*-----
SERIAL.C
void com_initialize (void);
void com_baudrate (
unsigned baudrate);
char com_putchar (
unsigned char c);
char com_gets (
char *z);
int com_getchar (void);
unsigned char com_rbuflen (void);
unsigned char com_tbuflen (void);
/*-----
TIMER.C
#define TIMERS0_TICKS_PER_SEC 100
void timer0_initialize (void);
  
```



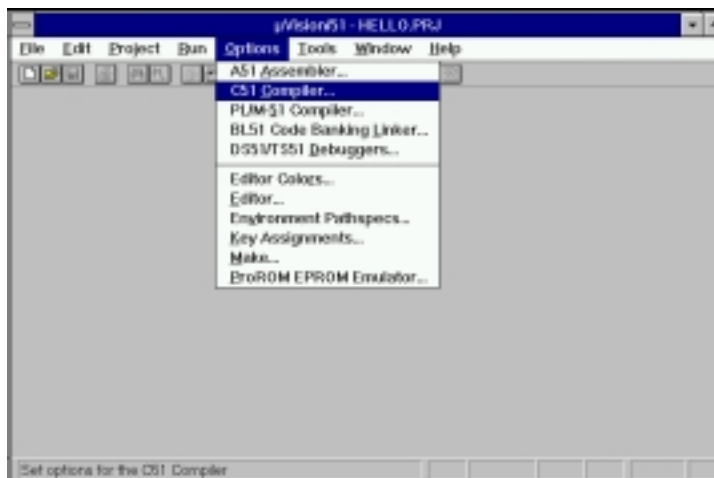
## Editor

$\mu$ Vision's built-in editor can be customized to emulate many popular text editors. You can change key assignments for almost all editor functions. The following table lists a few of the editor functions that are available:

Beginning of File	Destructive Backspace	Next Error
Beginning of Line	End of File	Open File
Beginning of Page	End of Line	Page Down
Cascade Windows	End of Page	Page Up
Close File	Exclusive Mark	Paste from Clipboard
Copy to Clipboard	Forward Quick Search	Previous Error
Cursor Down	Forward Replace	Previous Window
Cursor Left	Full Search	Print File
Cursor Right	Insert Template	Repeat Last Search
Cursor Up	Mark Block	Reverse Quick Search
Cut to Clipboard	Mark Columns	Undo
Delete	Mark Lines	Word Left
Delete Line	Move/Resize Window	Word Right
Delete to End of Line	New File	

## Menu Commands

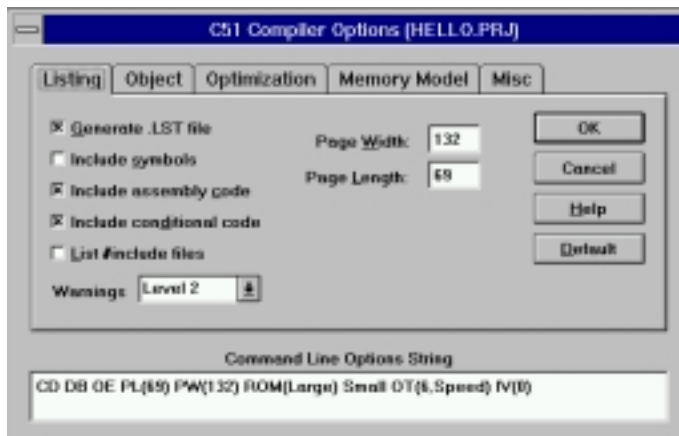
Through pull-down menus on the menu bar and editor commands, you control the  $\mu$ Vision operations. You may use either the mouse or the keyboard to access commands from the menu bar.



The menu bar provides you with access to menus for file operations, editor operations, project maintenance, external program execution (such as running the dScope debugger/simulator or another program), development tool option settings, window selection and manipulation, and on-line help.

## Development Tool Options

µVision lets you set options for software development tools such as the C51 compiler and A51 assembler. Simply select the appropriate item from the Options menu and use the mouse or the keyboard to change the options.



## 6

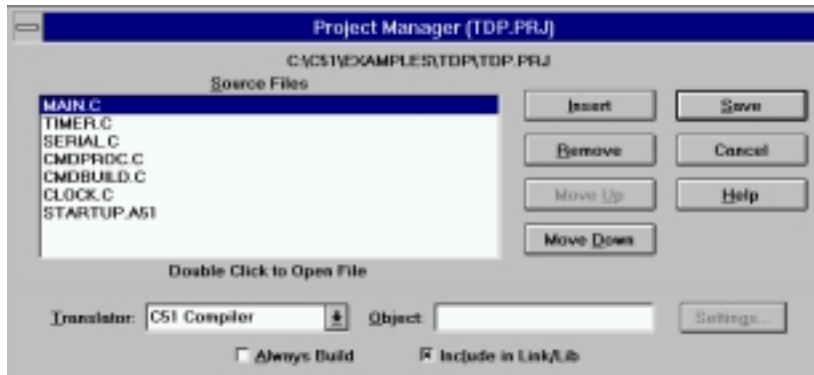
## Project Manager

Most embedded programs are composed of several source files. This means that a project includes a large collection of individual files. Some files may require compilation with the C51 compiler, some files may require assembly, and some files may require custom translation in order to create a target program.

To accommodate the intricacies of project maintenance, µVision includes a project manager facility. The project manager gives you a method of creating and maintaining a project so that the target program is always up-to-date. The project manager can easily handle file-to-file dependencies, including file nesting, as well as the exact sequence of operations required to build the target.

Use the project manager dialog box to define the source files that make up the project; use the make commands from the Project menu to compile source files

and to generate the target; then, use the simulator and emulator commands from the Run menu to execute, test, and debug your application.



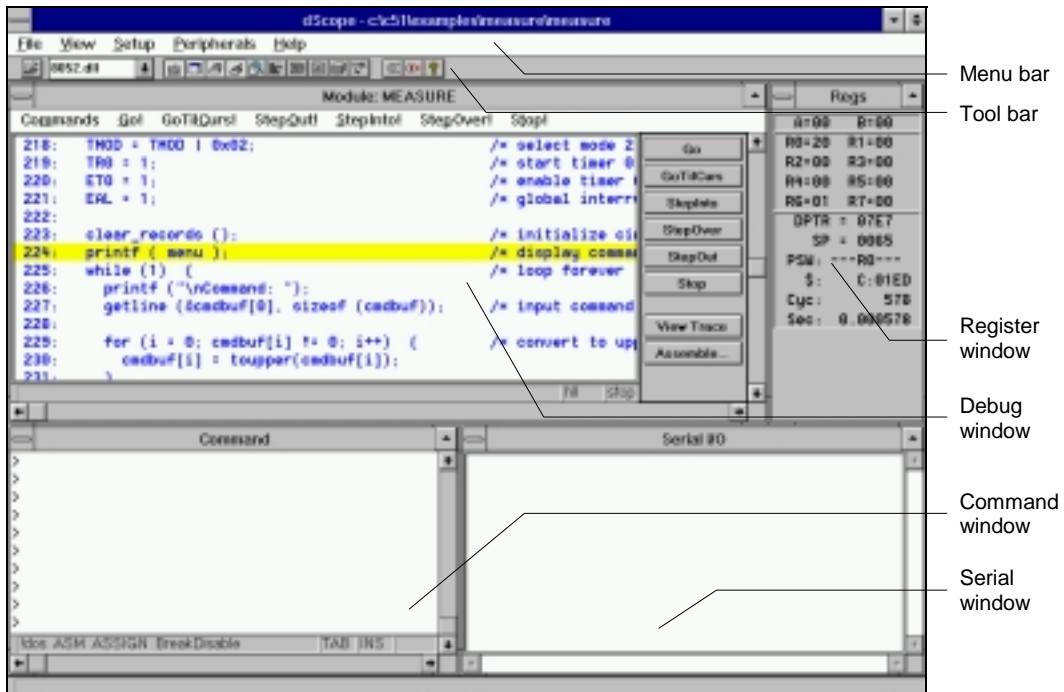
All aspects of a project are saved in a project file. The project file includes: the source files that make up the target program; the compiler, assembler, and linker command line options; the debugger and simulator options; and the make facility options.

## dScope Simulator/Debugger Overview

dScope is a source level debugger/simulator for the entire Keil Software product line. You can use dScope to debug the applications you develop using the C51 and C251 compilers and A51 and A251 assemblers. In addition, dScope lets you debug application written using the Intel PL/M-51 compiler and the ASM-51 assembler.

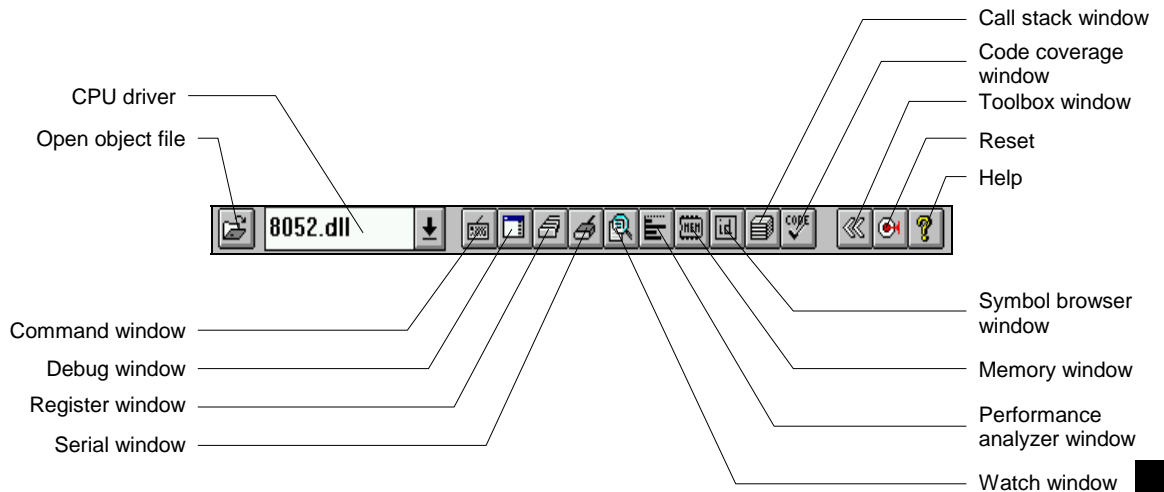
dScope is a software-only product that simulates most of the features of 8051 and 251 microcontrollers without actually having target hardware. You can use dScope to test and debug your embedded application before the hardware is ready. dScope simulates a wide variety of 8051 and 251 peripherals including the serial port, external I/O, and timers. Support for the various microcontroller derivatives is provided through the use of dynamic link libraries (DLLs).

In addition to simulating the CPU, dScope interfaces directly to the 8051 and 251 monitor programs and to several popular emulators.



## About the Debugger

In dScope, you may use the keyboard or the mouse to select menu commands, step through your application, and select debugging options. The dScope screen, pictured above, provides you with a menu bar for command entry, a tool bar where you can rapidly select command buttons, and several windows for displaying registers, memory contents, serial I/O, and commands. You can quickly display and hide the windows shown above with the buttons on the tool bar.



## CPU Simulation

dScope simulates virtually every derivative of the 8051 and 251 microcontrollers. Support for each CPU is provided through the use of DLLs. Before you load your target application, you must select the appropriate CPU driver from the CPU driver drop-down box on the tool bar. You may also select the Load CPU driver command from the File menu. The following CPU drivers are included with dScope.

CPU Driver DLLs	Supported Derivatives
<b>80251S.DLL</b>	Intel 80C251SA, 80C251SB, 80C251SQ, and 80C251SP
<b>82930.DLL</b>	Intel 82930 USB
<b>MON51.DLL</b>	Keil Software 8051 Target Monitor
<b>MON251.DLL</b>	Keil Software 251 Target Monitor
<b>RISM251.DLL</b>	Intel Reduced Instruction Set Monitor (RISM) for the 251
<b>80320.DLL</b>	Dallas Semiconductor 80C320, 80C520, and 80C530
<b>8051.DLL</b>	8031, 8051, 80C31, and 80C51

CPU Driver DLLs	Supported Derivatives
80515.DLL	80C515 and 80C535
80515A.DLL	80C515A and 80C535A
80517.DLL	80C517 and 80C537
80517A.DLL	80C517A and 80C537A
8051FX.DLL	8051FA, 8051FB, and 8051FC
8052.DLL	8032, 8052, 80C32, and 80C52
80552.DLL	8xC552
80751.DLL	8xC750, 8xC751, and 8xC752
80410.DLL	8xCL410
80781.DLL	8xCL781

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

In addition to memory mapping, dScope also provides support for the integrated peripherals of the various 8051 and 251 derivatives. The CPU's on-chip peripherals are supported by the CPU driver in the DLL.

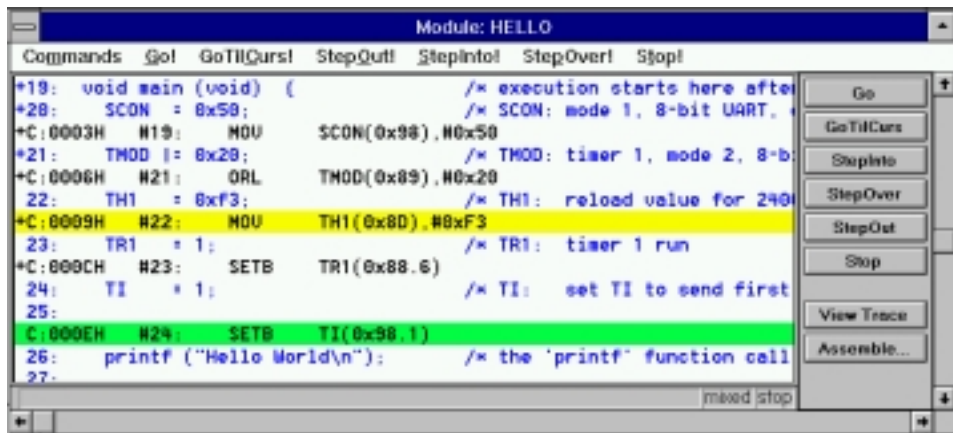
You can select and display the on-chip peripheral components using the Peripherals menu. You can also change the aspects of each peripheral using the controls in the dialog boxes.



## The Debug Window

After you have loaded the appropriate CPU driver, you are ready to load your target program. You can use the button on the tool bar to open your object file, or you can use the Load object file command from the File menu.

Once your application is loaded, the dScope debug window displays your C, assembly, or PL/M-51 source text.



Three display formats are available from the Command menu in the debug window. They are:

- **View High Level.** This display format shows your original source text exactly as it appears in your source files.
- **View Mixed.** This display format shows your original source text mixed with the assembly code generated by the compiler or assembler.
- **View Assembly.** This display format shows only the assembly code generated for your source.

In addition to target program, the debug window can display a trace history of up to 512 previously executed instructions. To enable the trace history, select the Record Trace command from the Command menu in the debug window.

## Command Window

You interact with dScope by entering commands from the keyboard and selecting options with the mouse. You can enter nearly all dScope commands in the command window. dScope responds to the commands you enter at the command prompt ('>').



You can interactively display and change variables, registers, and memory locations from the command window. You can also enter assembly code to patch or test parts of your program.

For example, you can type the following text commands at the command prompt:

6

Text	Effect
<b>DPTR</b>	Display the DPTR register.
<b>R7 = 12</b>	Assign the value 12 to register R7.
<b>time.hour</b>	Displays the hour member of the time structure.
<b>time.hour++</b>	Increments the hour member of the time structure.
<b>index = 0</b>	Assigns the value 0 to index.

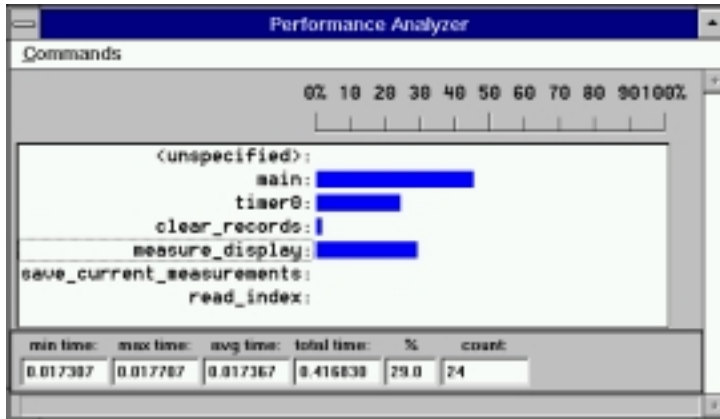
You are not limited to using the command window to control dScope. You can also use the mouse to select pull-down menus from the menu bar and invoke commands from the tool bar.





## Performance Analyzer Window

dScope has a built-in performance analyzer that lets you record timing statistics for functions and program blocks. Performance analysis results are displayed in the performance analyzer window.



The performance analyzer window shows the name of each function or memory range of each block along with a bar graph showing the percentage of time spent in that function or block. You may select a function to view statistics in the bottom portion of the performance analyzer window. The following statistics are maintained for each function or program block:

6

- **min time**            Minimum time spent in the function or block,
- **max time**            Maximum time spent in the function or block,
- **avg time**             Average amount time spent in the function or block,
- **total time**            Total time spent in the function or block,
- **count**                Number of times the function or block was entered.

## Other Features

In addition to the features described above, dScope offers numerous other functions that provide a robust debugging environment.

### Functions

A powerful feature of dScope is its ability to let you define and use C-like functions for a wide variety of applications. For example, you can create dScope functions to manipulate the on-chip peripherals, extend the command set of dScope, and generate digital and analog input to hardware ports. There are three types of functions available to dScope:

- **User Functions** extend the command scope of the debugger,
- **Signal Functions** generate input to the 8051 peripherals,
- **Built-in Functions** provide convenient utility routines (like **printf** and **memset**) that you can use in user or signal functions.

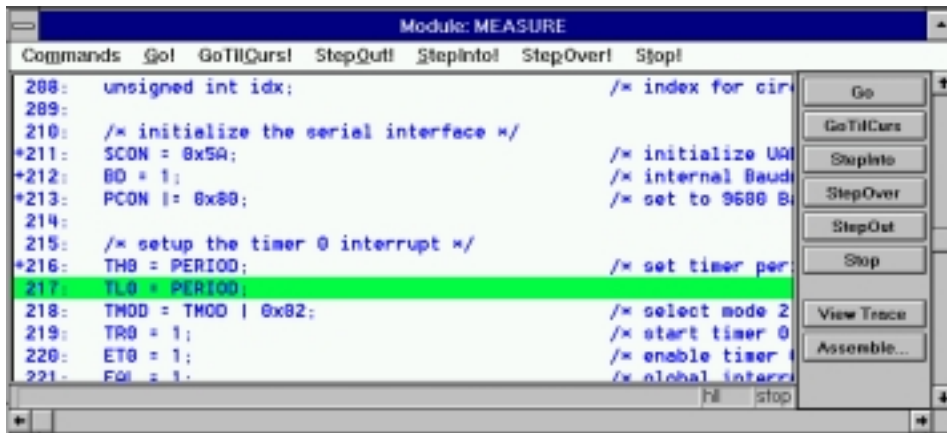
Refer to “Signal Functions” on page 99 for an example of how to use functions in dScope.

### Breakpoints

It is easy to set breakpoints on high-level statements, assembler instructions, and conditional expressions. Simply move the mouse pointer to the line or instruction and double-click. You can even set a breakpoint based on the type of memory access type or repetition factor. When dScope reaches a breakpoint, it can perform a wide range of operations—from simple probing to running macro functions.

## Code Coverage

dScope provides a code coverage function which marks the lines of code that have been executed. In the debug window, lines of code which have been executed are marked with a plus sign ('+') in the left column.



```
Module: MEASURE
Commands  Go!  GoTo/Curs!  StepOut!  StepInto!  StepOver!  Stop!

208:  unsigned int idx;                                /* index for cir
209:
210:  /* initialize the serial interface */
+211:  SCON = 0x5A;                                    /* initialize UA
+212:  BD = 1;                                        /* internal Baud
+213:  PCON |= 0x80;                                  /* set to 9600 B
214:
215:  /* setup the timer 0 interrupt */
+216:  TH0 = PERIOD;                                  /* set timer per
217:  TL0 = PERIOD;
218:  TMOD = TMOD | 0x82;                              /* select mode 2
219:  TR0 = 1;                                        /* start timer 0
220:  ET0 = 1;                                        /* enable timer
221:  EA = 1;                                          /* global interri
```

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 box also provides useful information and code coverage statistics.

## Sample Programs

This section describes the sample programs that are included in our evaluation kits and product 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 `\C51\EXAMPLES\` directory. Each sample program is stored in a separate subdirectory along with project files and batch files that help you quickly build and evaluate each sample program.

The following table lists the sample programs and their directories.

Directory	Description
<code>\A51\</code>	A51 is a sample program for the A51 assembler.
<code>\BADCODE\</code>	<b>BADCODE</b> is a sample program with a number of syntax errors. Use $\mu$ Vision to open the BADCODE.PRJ project file and compile. $\mu$ Vision takes you to each error in BADCODE.C. Refer to "BADCODE: An Example with Syntax Errors" on page 105 for more information about this sample program.
<code>\BL51_EX1\</code>	<b>BL51_EX1</b> demonstrates a bank switching application written in C. This sample program invokes functions in different code banks. Build this program using the BL51_EX1.PRJ project file.
<code>\BL51_EX2\</code>	<b>BL51_EX2</b> demonstrates a C program that has constant messages stored in different code banks. Build this program using the BL51_EX2.PRJ project file.
<code>\BL51_EX3\</code>	<b>BL51_EX3</b> demonstrates a bank switching program that has only one module with functions located in different banks. Build this program using the BL51_EX3.PRJ project file.
<code>\BL51_EX4\</code>	<b>BL51_EX4</b> demonstrates a bank switching, Intel PL/M-51 program that calls functions in different code banks. This program is the PL/M-51 equivalent to <b>BL51_EX1</b> . Build this program using the BL51_EX4.PRJ project file. The Intel PL/M-51 compiler is required.
<code>\CSAMPLE\</code>	The <b>CSAMPLE</b> sample program demonstrates a simple addition and subtraction calculator. This sample program is a multiple module project that you can build using the CSAMPLE.PRJ project file.
<code>\DHRY\</code>	The <b>DHRY</b> example is a DHRYSTONE benchmark program that calculates and displays the dhrystones per second for the host CPU. This example is mainly provided for benchmark enthusiasts. Build this program using the DHRY.PRJ project file.
<code>\FIB\</code>	The <b>FIB</b> sample program generates fibonacci numbers and shows you how to use the reentrant function attribute to declare recursive functions. Build this sample program using the FIB.PRJ project file.
<code>\HELLO\</code>	The <b>HELLO</b> sample program is the embedded 8051 C Hello World program. Use the HELLO.PRJ project file to build this program. Refer to "HELLO: Your First 8051/251 C Program" on page 81 for more information about this sample program.
<code>\LSIEVE\</code>	<b>LSIEVE</b> demonstrates the large model version of the sieve of Eratosthenes prime number generator. This example is mainly provided for benchmark enthusiasts. Build this program using the LSIEVE.PRJ project file.

Directory	Description
\MEASURE\	The <b>MEASURE</b> sample C program collects analog and digital data. It simulates a data acquisition system that might be found in a weather station or in a process control application. Build this program using the MEASURE.PRJ project file. Refer to “MEASURE: A Remote Measurement System” on page 88 for more information about this sample program.
\RTX_EX1\	The <b>RTX_EX1</b> sample program demonstrates round-robin multitasking using RTX-51 Tiny. Build this program using the RTX_EX1.PRJ project file.
\RTX_EX2\	The <b>RTX_EX2</b> sample program demonstrates an RTX-51 Tiny application that uses signals. Build this program using the RTX_EX2.PRJ project file.
\SAMPL517\	The <b>SAMPL517</b> sample program provides an RPN-style calculator that takes advantage of the 80C517 arithmetic processor. Build this program using the SAMPL517.PRJ project file.
\SSIEVE\	The <b>SSIEVE</b> sample program demonstrates the small model version of the sieve of Eratosthenes prime number generator. This example is mainly provided for benchmark enthusiasts. Build this program using the SSIEVE.PRJ project file.
\TDP\	The <b>TDP</b> sample program demonstrates how to use interrupt-driven serial I/O to interface to an alarm clock driven by an interrupt-driven timer. Build this program using the TDP.PRJ project file.
\TRAFFIC\	The <b>TRAFFIC</b> sample program shows how to control a traffic light using the RTX-51 Tiny real-time executive. Build this program using the TRAFFIC.PRJ project file.
\WHETS\	The <b>WHETS</b> example is a WHETSTONE benchmark program that calculates and displays the number whetstones per second for the host CPU. This example is mainly provided for benchmark enthusiasts. Build this program using the WHETS.PRJ project file.

## 6

To begin using one of the sample files, you must switch to the directory in which the sample resides. Then, you may use either the provided DOS batch files or the  $\mu$ Vision for Windows project file to build and test the sample program.

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
- BADCODE: An Example with Syntax Errors

## HELLO: Your First 8051/251 C Program

The HELLO sample program is located in the `\C51\EXAMPLES\HELLO\` directory. 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`, which is listed below.

```

/*-----
HELLO.C

Copyright 1995 KEIL Software, Inc.
-----*/

#pragma DEBUG OBJECTTEXTEND CODE /* pragma lines can contain */
/* command line directives */

#include <reg51.h> /* special function register declarations */
/* for the intended 8051 derivative */

#include <stdio.h> /* prototype declarations for I/O functions */

/*****
/* main program */
/*****
void main (void) { /* execution starts here after stack init */
    SCON = 0x50; /* SCON: mode 1, 8-bit UART, enable rcvr */
    TMOD |= 0x20; /* TMOD: timer 1, mode 2, 8-bit reload */
    TH1 = 0xf3; /* TH1: reload value for 2400 baud */
    TR1 = 1; /* TR1: timer 1 run */
    TI = 1; /* TI: set TI to send first char of UART */

    printf ("Hello World\n"); /* the 'printf' function call */

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

```

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.

## Hardware Requirements

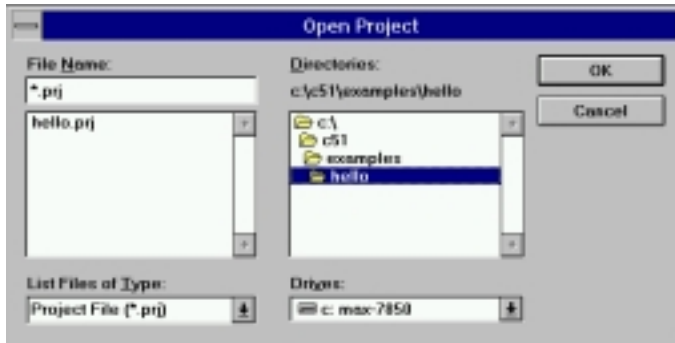
The hardware for HELLO is based on the standard 8051CPU. The only on-chip peripheral used is the serial port. You do not actually need a target CPU because dScope lets you simulate the hardware required for this program.

## HELLO Project File

In  $\mu$ Vision, applications are maintained in a project file. The project file contains names of all source files associated with the project and also tells the

tools how to compile, assemble, and link to generate an executable target program.

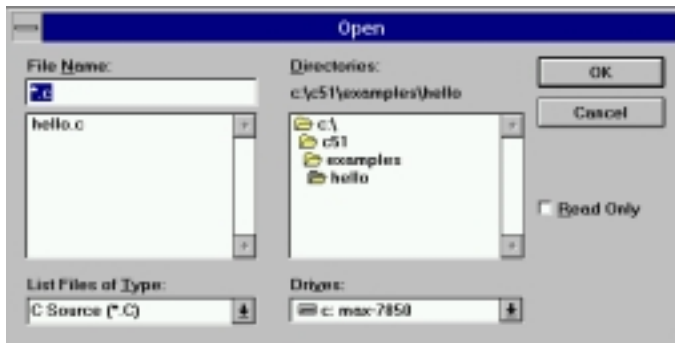
A project file, called **HELLO.PRJ**, has been created for **HELLO**. To load this project file, select the Open command from the Project menu and open the **HELLO.PRJ** project file from the **\C51\EXAMPLES\HELLO** directory.



## Editing HELLO.C

You can now edit **HELLO.C**. Select the Open command from the File menu.  $\mu$ Vision prompts you with the Open File dialog box. Select **HELLO.C** from the files list and select the OK button.

6





$\mu$ Vision loads and displays the contents of `HELLO.C` in a window.

```

-----
HELLO.C
Copyright 1995 KEEL Software, Inc.
-----*/

#pragma DEBUG OBJECTEND CODE /* pragma lines can contain state C51 */
/* command line directives */

#include <reg51.h> /* special function register declarations */
/* for the intended 8051 derivative */

#include <stdio.h> /* prototype declarations for I/O functions */

/*****
/* main program */
/*****
void main (void) { /* execution starts here after stack init */
    SCON = 0x50; /* SCON: mode 1, 8-bit USART, enable rcvr */
    TMO0 = 0x20; /* TMO0: timer 1, mode 2, 8-bit reload */
    TH1 = 0xF3; /* TH1: reload value for 2400 baud */
    TM1 = 1; /* TM1: timer 1 run */
    TI = 1; /* TI: set TI to send first char of USART */

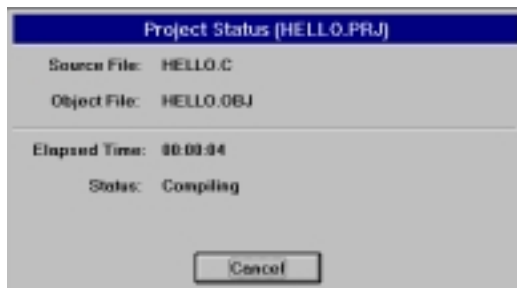
    printf ("Hello World!\n"); /* the 'printf' function call */

    while (1) { /* An embedded program does not stop and */
        /* never returns. We've used an endless */
        /* loop. You may wish to put in your own */
        /* code were we've printed the data (...). */
    }
}

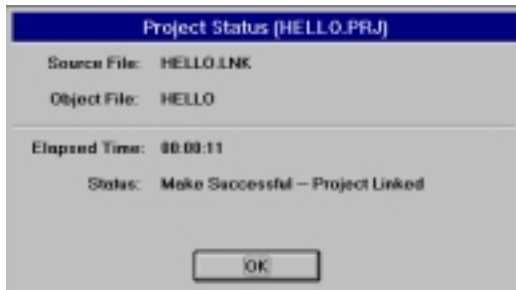
```

## Compiling and Linking HELLO

When you are ready to compile and link your project, click on the Build All button on the tool bar or select the Make: Build Project command from the Project menu.  $\mu$ Vision begins to compile and link the source files in your project and create an absolute object module that you can load into dScope for testing. During the build,  $\mu$ Vision displays the status in a window.



When the build is complete,  $\mu$ Vision displays a message indicating the build is finished.



You may press **Esc** at any time to halt the build.

---

**NOTE**

*You should encounter no errors when you use  $\mu$ Vision with the provided sample projects. If  $\mu$ Vision says it cannot find or run the compiler or linker, check your **PATH** for the `\C51\BIN` directory. If it is not there, you must add it so that  $\mu$ Vision can find the compiler and the other tools. You can add the path specifications in  $\mu$ Vision when you select the *Environment Paths* command from the *Options* menu.*

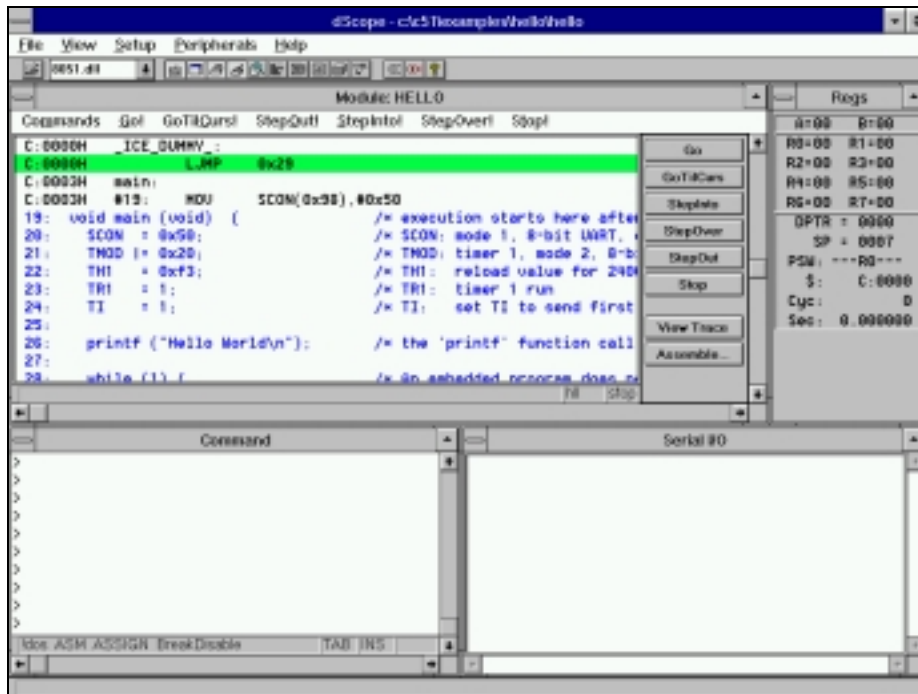
---

## Testing HELLO With dScope

Once the HELLO program is compiled and linked, you can test it with the dScope debugger/simulator. In  $\mu$ Vision, select the DS51 Simulator command from the Run menu and press **Enter** when the dScope Command Arguments dialog box displays.

$\mu$ Vision passes an initialization file (**HELLO.INI**) to dScope. This file contains commands for dScope that load the CPU driver DLL and the HELLO sample program.

When dScope loads, the following screen displays.



### NOTE

The first time you invoke dScope, you may need to change the fonts and colors used for the different windows. Select the Colors and Fonts command from the Setup menu to configure the different windows in dScope.

## Running HELLO

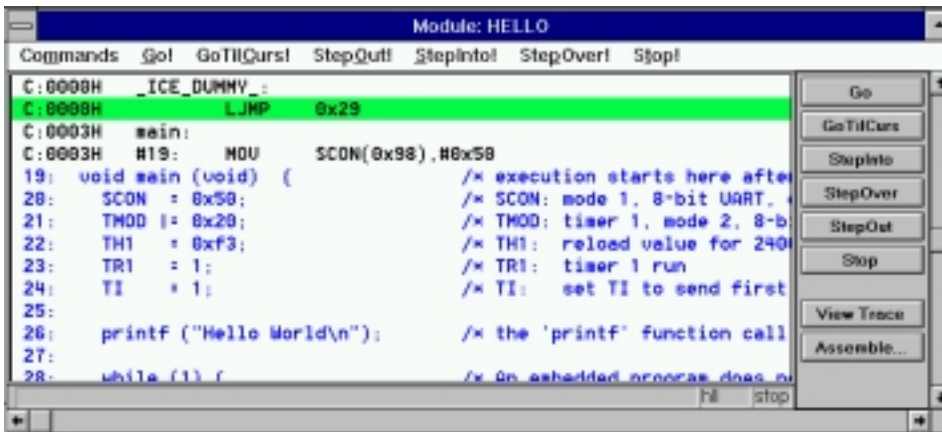
To run the HELLO program, click on the Go button in the debug window or enter **g** at the command prompt. The HELLO program executes and displays the text “Hello World” in the serial window.



After HELLO outputs “Hello World,” it begins executing an endless loop. To halt execution, click on the Stop button in the debug window or type **Ctrl+C**. After you have halted program execution, you may type **exit** to leave the dScope debugger.

## Single-Stepping Through HELLO

You can single-step through the HELLO program using the Step buttons in the debug window.



First, make sure to reset the CPU driver. To do this, make sure program simulation is halted, then type the following lines at the command prompt:

```
reset  
g,main
```

The `reset` command resets the simulated 8051 CPU. The `g,main` command begins executing the program and stops when it reaches the main C function.

To step through the HELLO program, click on the StepOver button in the debug window. Each time you click on this button, the simulator executes one statement. The current instruction is always highlighted, but the highlight moves each time you step. You may continue stepping through your program by clicking on the StepOver button.

You may exit dScope at any time. To do so, halt execution of HELLO and enter `exit` at the command prompt.

## MEASURE: A Remote Measurement System

The MEASURE sample program is located in the `\C51\EXAMPLES\MEASURE\` directory. 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`.

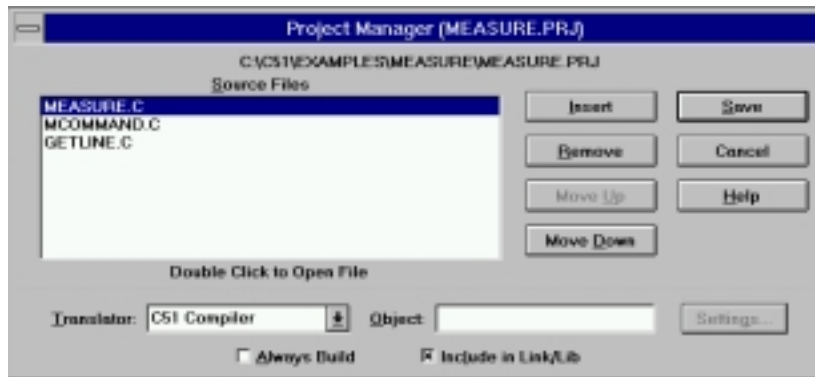
MEASURE records data from two 8-bit digital ports and four 8-bit analog-to-digital inputs. A timer controls the sample rate. The sample interval can be configured from 1 millisecond to 60 minutes. Each measurement saves the current time and all of the input channels to an 8 Kbyte RAM buffer.

### Hardware Requirements

The hardware for MEASURE is based on the 80517 CPU. This microcontroller provides analog and digital input capability. Port 4 and port 5 are used for the digital inputs and AN0 through AN3 are used for the analog inputs. You do not actually need a target CPU because dScope lets you simulate all the hardware required for this program.

## MEASURE Project File

The project file for the MEASURE sample program is called `MEASURE.PRJ`. To load this project file, select the Open command from the Project menu and open `MEASURE.PRJ` from the `\C51\EXAMPLES\MEASURE` directory. Select the Edit Project command from the Project menu to display the Project Manager dialog box.



The Project Manager dialog box shows the source files that compose the MEASURE project. There are three source files in this project.

- |                   |  |
|-------------------|--|
| <b>MEASURE.C</b>  | This source file contains the main C function for the measurement system and the interrupt routine for timer 0. The main function initializes all peripherals of the 80517 and performs command processing for the system. The timer interrupt routine, <code>timer0</code> , manages the real-time clock and the measurement sampling of the system. Timer 0 was used to maintain compatibility with the 8051 which can be used if fewer input channels are required. |
| <b>MCOMMAND.C</b> | This source file 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.   |
| <b>GETLINE.C</b>  | This source file contains the command-line editor for characters received from the serial port.  |

To open a source file from the Project Manager dialog box, double-click on the filename. To close the Project Manager dialog box, press **Esc** or click on the Cancel button.

## Compiling and Linking MEASURE

When you are ready to compile and link MEASURE, click on the Build All button on the tool bar or select the Make: Build Project command from the Project menu.  $\mu$ Vision begins to compile and link the source files in MEASURE and displays a message when the build is finished.

Once compiling and linking are complete, you are ready to begin testing the MEASURE sample program.

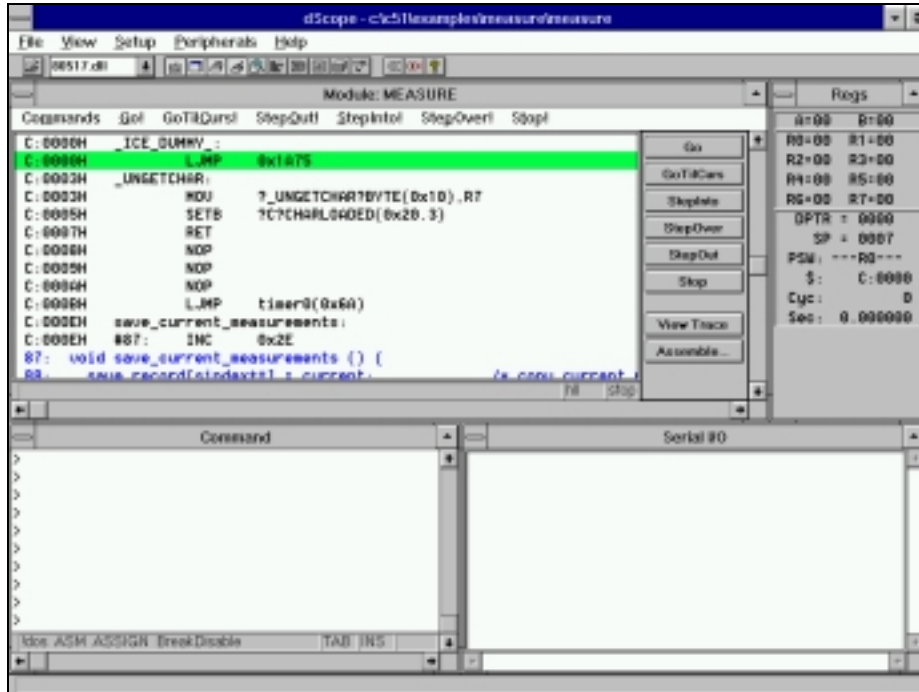
## Testing MEASURE With dScope

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 host computer or dumb terminal to communicate with the 80517 CPU. If you do not have target hardware, you can use dScope to simulate the hardware. You can also use the serial window in dScope to provide serial input.

Once the MEASURE program is compiled and linked, you can test it with dScope. In  $\mu$ Vision, select the DS51 Simulator command from the Run menu and press **Enter** when the dScope Command Arguments dialog box displays.



The initialization file that  $\mu$ Vision passes to dScope automatically loads the CPU driver and MEASURE program. Once these are loaded, dScope displays the following screen.



## Remote Measurement System Commands

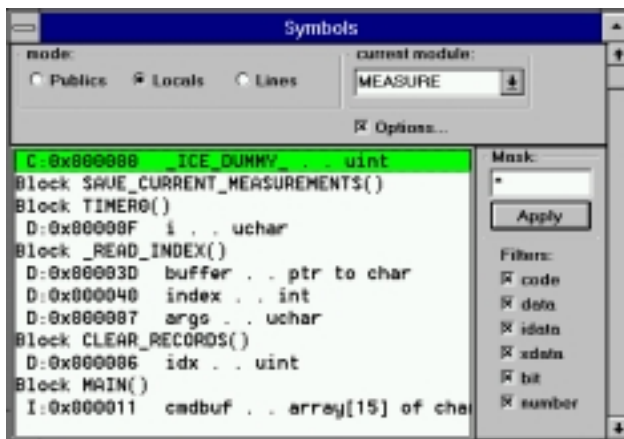
The serial commands that MEASURE supports are listed in the following table. These commands are composed of ASCII text characters. All commands must be terminated with a carriage return.

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.tt</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.

Command	Serial Text	Description
Read	R [count]	Displays the recorded measurements. You may specify the number of most recent samples to display with the read command. If no count is specified, the read command transmits all recorded measurements. You can read measurements on the fly if the interval time is more than 1 second. Otherwise, the recording must be stopped.
Quit	Q	Quits the measurement recording.

## Viewing Debug Symbols

The MEASURE sample program is configured for full debug information and includes public and local symbols, line numbers, and high-level type information. To view this information, click on the Symbol Browser button on the tool bar to open the symbol browser window. Then, select the Locals radio button and the Options check box as shown below.



dScope supports the drag and drop feature of Windows and lets you access the symbols this way. Use the mouse to drag and drop the **idx** symbol from the symbol browser window to the command window. The fully qualified symbol name with module name and function name are inserted as shown. The qualifiers are separated by the backslash character ('\'). Select the command window and press **Enter**. dScope displays the value of **idx**.

You may filter the symbols displayed by selecting the memory space filter. If you clear the data check box, all symbols in the data memory area are removed from the display.

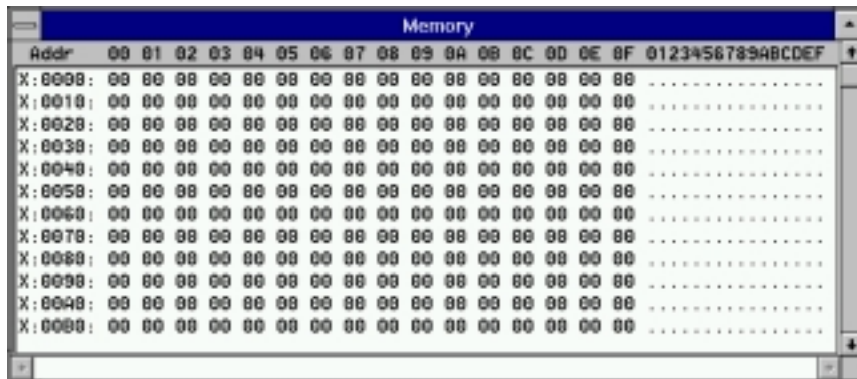
You can specify a search mask to limit the symbols displayed. To limit the symbol list to those beginning with the letter I, enter “I\*” and click the Apply button.

## Viewing Memory Contents

dScope displays memory in HEX and ASCII in the memory window. Open the memory window by clicking on the Memory button on the tool bar. In the command window, enter the address range you want to view, for example:

```
D X:0x0000, X:0xFFFF
```

Since the memory window cannot show the entire memory range at once, you may use the scroll bars to scroll through the memory area. The bounds for scrolling are defined by the address range specified, 0x0000 to 0xFFFF for this example.



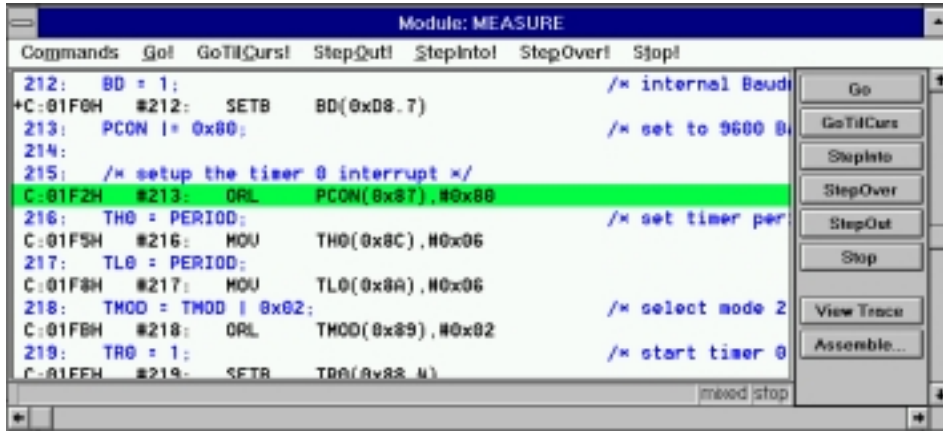
To display the on-chip data memory, enter the following in the command window.

```
D I:0x0000, I:0xFF
```

dScope can dynamically update the memory window while your application is running. To toggle dynamic updating, select the Update Memory window command from the Setup menu. When Update Memory window is checked, dynamic updating is enabled.

## Changing the View Mode

dScope lets you change the view mode in the debug window. Display the debug window using the debug button on the tool bar. Then, to change the view mode, open the Commands menu in the debug window and select View High level, View Mixed, or View Assembly. For example, View Mixed changes to the mixed source and assembly display.



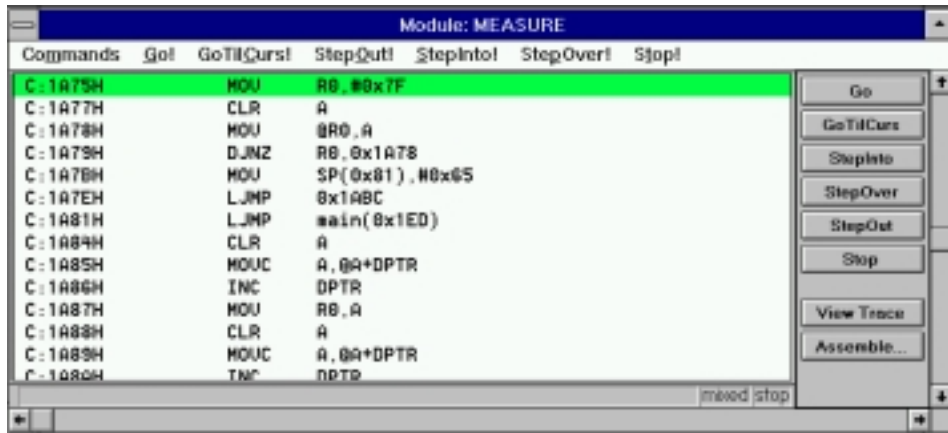
The debug window shows intermixed source and assembly lines.

## 6

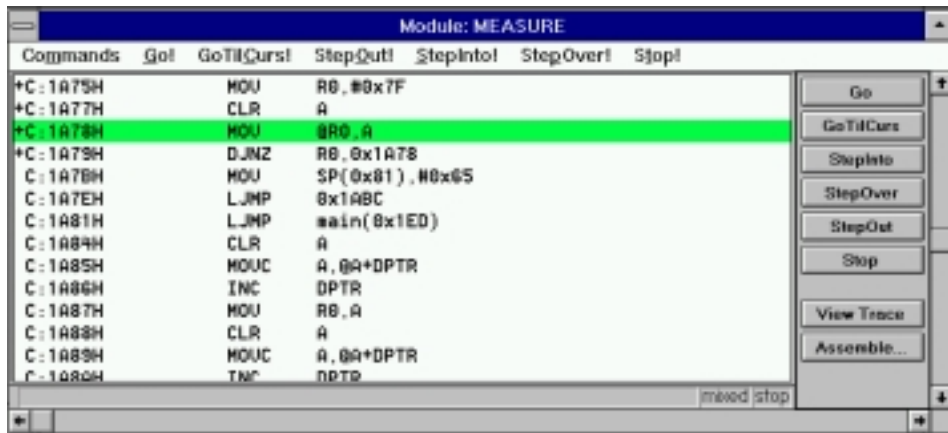
### Program Execution

Before you begin simulating the MEASURE program, use the Debug, Register, and Serial buttons on the tool bar to display the debug, register, and serial windows. You may disable other windows if your screen is not large enough.

From the toolbar, select the reset button to reset dScope. In the debug window, select the View Mixed command from the Commands menu. Then, click on the StepInto button once.



The StepInto button lets you single-step through your application and into function calls. Click on the StepInto button a few more times to get to the loop which clears the on-chip data space of the CPU.



To skip the initialization code and go directly to the main function, select the command window and enter “G,main”. dScope executes the startup code and halts on the first statement in the main function.

## Go Until Current Cursor Line

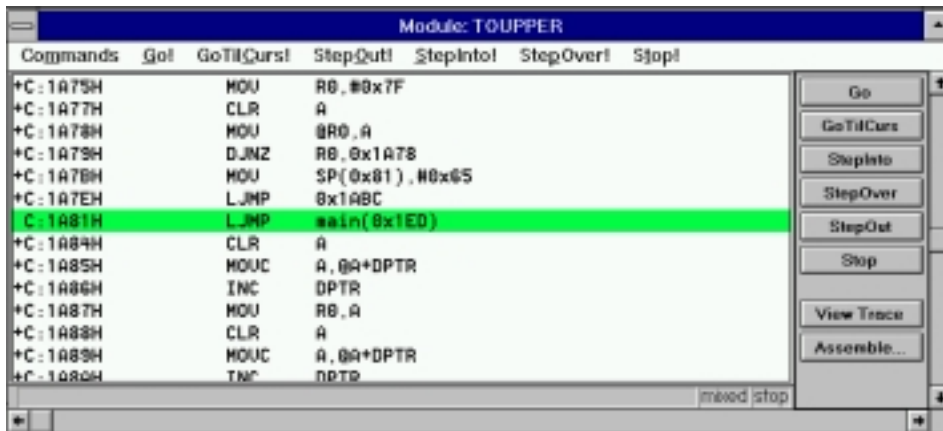
The current cursor line is the line which marks the current assembly or high-level statement. You can move the line using the keyboard or the mouse.

dScope lets you use the current cursor line as a temporary breakpoint. Use this feature to skip over code in your application. For example, you can skip over the

initialization code and stop one instruction before the main function is called. You can do this in one of two ways:

- **Variation 1:** Move the cursor line to the **LJMP main** instruction. You can use the cursor keys or you can click the mouse on that line. Click on the GoTilCurs button in the debug window. dScope starts execution at the current program counter and stops at the current cursor line.
- **Variation 2:** Double-click, with the right mouse button, on the **LJMP main** instruction. This makes the selected line the current cursor line, starts execution from the current PC, and stops when the current line is reached.

The program counter is now at the **LJMP main** instruction.

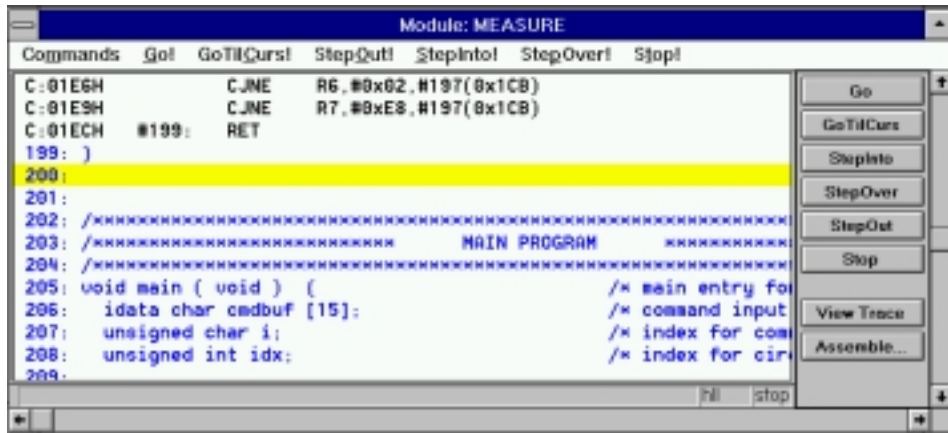


### NOTE

After performing this command, the current cursor line and the current program counter (PC) line are the same. The background color used for the line is the PC highlight color.

## Stepping Through High-Level Statements

Click on the StepInto button in the debug window and dScope jumps to the main function of the MEASURE sample program. Select the View High level command from the debug window Commands menu.



When viewing your application in high level mode, the meaning of a step changes to mean one high-level statement instead of one assembly instruction. Click on the StepInto button and watch as the current program counter line moves down the screen.

#### **NOTE**

*The StepOver button operates much like the StepInto button with the exception that a function call is considered a single statement.*

## **Stepping Out of a Function**

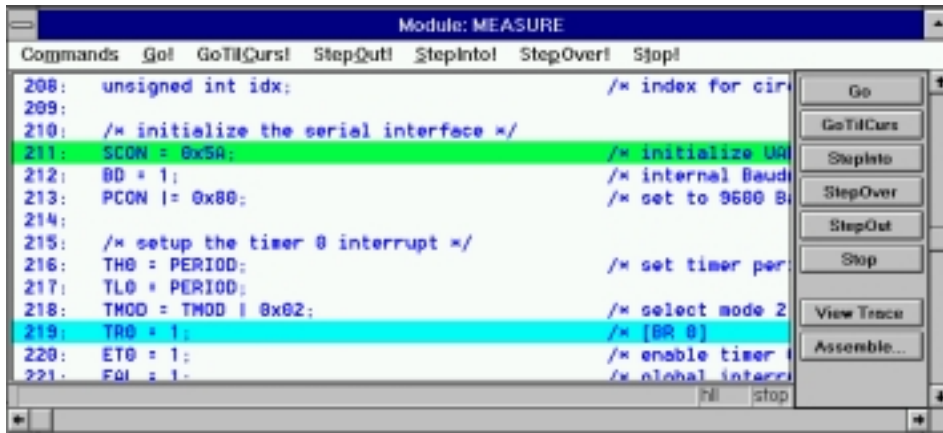
On occasion, you may accidentally step into a function unnecessarily. You can use the StepOut button to complete execution of that function and return to the statement immediately following the function call.

#### **NOTE**

*You cannot StepOut from the main function because it is invoked by a long jmp (LJMP) rather than a call instruction.*

## **Setting and Removing Breakpoints**

You can set an execution breakpoint in the debug window by double-clicking on the desired source line. The selected line is highlighted and a [BR n] label is displayed at the end of the line. If we set a breakpoint on the TR0 = 1 statement, the debug window appears as follows:

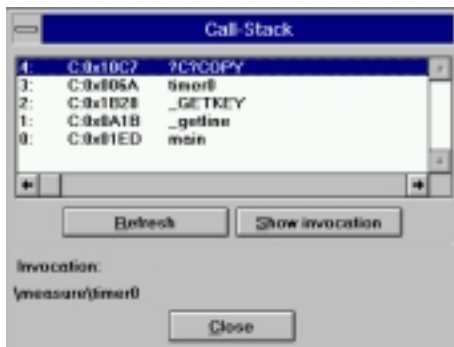


Click on the Go button and dScope starts execution from the current program counter and stops when the breakpoint is reached. To remove a breakpoint, double-click on the line containing the breakpoint.

## Call Stack

dScope internally tracks function nesting as the program executes. You can view the function nesting at any time by opening the Call Stack window. Use the Call Stack button on the tool bar to display Call Stack window.

# 6



This dialog box lists all currently nested functions. Each line contains a nesting level number, the numeric address of the invoked function, and the symbolic name of the function if debug information is available.

You can display the caller of a function by selecting the function from the list. Then, you can use the Show invocation button to display the function call in the debug window.



## Port Inputs

dScope provides two different ways to set digital and analog port inputs. You can use the Peripheral menu in the main window to view and change the status of input lines or you can enter I/O values in the command window. The following commands change port values in the command window.

```
PORT4=0x23          set digital input PORT3 to 0x23.
AIN1=3.3           set analog input AIN1 to 3.3 volts.
```

## Signal Functions

dScope lets you create signal functions to provide an input signal for digital or analog inputs. To load a signal function, halt program execution by clicking on the Stop button in the debug window and enter the following command in the command window.

```
INCLUDE analog.inc
```

This loads the analog function from the file ANALOG.INC. This file defines a signal function that adjusts the analog value that appears on analog channel 0. This function appears as follows.

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

    printf ("ANALOGO (%f) ENTERED\n", limit);
    while (1) {                                /* forever */
        volts = 0;
        while (volts <= limit) {
            ain0 = volts;                       /* analog input-0 */
            twatch (30000);                     /* 30000 Cycles Time-Break */
            volts += 0.5;                       /* increase voltage */
        }

        volts = limit-0.5;
        while (volts >= 0.5) {
            ain0 = volts;
            twatch (30000);                     /* 30000 Cycles Time-Break */
            volts -= 0.5;                       /* decrease voltage */
        }
    }
}
```

After loading the analog include file, enter the following commands in the command window.

```
ANALOGO (5.0)
G
```

These commands set the limit for analog channel 0 to 5.0 volts and start program execution.

Select the serial window and type **D Enter**. You should see the analog channel 0 signal begin swinging from 0 to 5 volts.

## Trace Recording

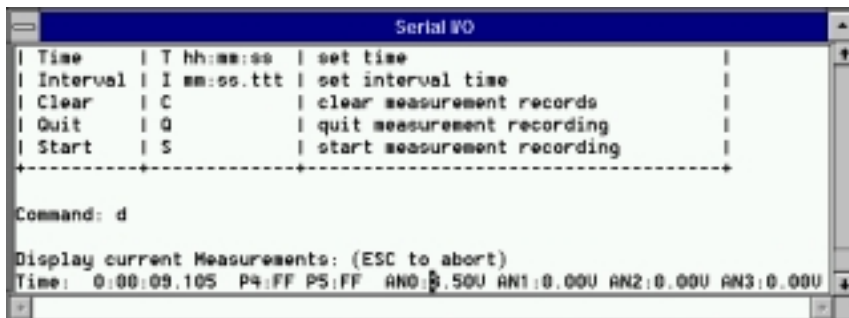
It is common during debugging to reach a breakpoint where you require information like register values and other circumstances which led to the breakpoint. dScope provides trace recording for this purpose.

To enable trace recording, select the Record trace command from the Commands menu to toggle instruction trace recording. When trace recording is enabled, dScope records up to 512 assembly instructions and register contents.

You can use trace recording with the MEASURE example. Start running the MEASURE program (click on the Go button in the debug window) and select the serial window. MEASURE displays a menu and waits for input after displaying **command**. In the serial window, enter **d**.

When you enter this command, MEASURE begins to display measurement values, the record time, two port values, and finally the analog input values.

6



```

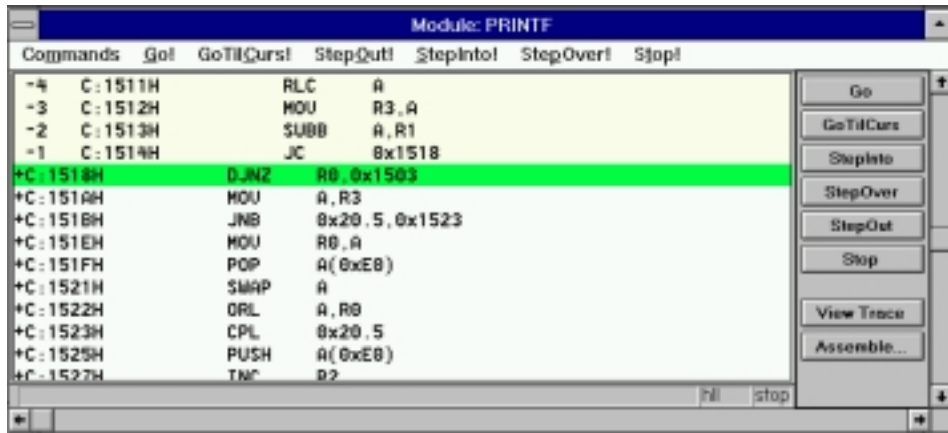
Serial I/O
| Time      | T hh:mm:ss | set time      |
| Interval  | I mm:ss.ttt | set interval time |
| Clear     | C           | clear measurement records |
| Quit      | Q           | quit measurement recording |
| Start     | S           | start measurement recording |
+-----+-----+-----+
Command: d

Display current Measurements: (ESC to abort)
Time: 0:00:09.105 P4:FF P5:FF AN0:5.500 AN1:0.000 AN2:0.000 AN3:0.000

```

The serial window displays what you would see on a dumb terminal connected to the 80517's serial port.

Click on the Stop button in the debug window. This halts program execution immediately. Click on the View Trace button to view the trace buffer.



The upper portion of the debug window shows the trace history. The lower portion of the debug window shows instructions from the program counter. The program counter line is the delimiter between the trace history and instructions not yet executed.

The trace history lines begin with negative numbers. The newest trace buffer entry is  $-1$ . The oldest entry is  $-511$ . When the buffer overflows, the oldest entries are removed to make space for new entries.

You may scroll into the trace buffer using the keyboard or the mouse. The register window shows the register contents for the selected instruction in the trace buffer.

---

### **NOTE**

*Program execution must be stopped before you can view the trace buffer.*

---

## **Watchpoints**

Watchpoints are used to view the contents of simple variables, structures, and arrays. You may setup watchpoints using the Watchpoints dialog box. To display this dialog box, select the Watchpoints command from the Setup menu.

The following steps show you how to define two watchpoints: one for the variable **sindex** which is an unsigned int and one for the structure **current** which contains a nested **time** struct.

To add a watchpoint for **sindex**:  
Type **sindex** in the Expr input line and click on the Define watch button.

To add a watchpoint for **current**:  
Type **current** in the Expr input line, select the Multiple radio button to display structure members on separate lines, and click on the Define watch button.



The watch window now contains the two watch expressions just defined.

The first watch expression shows the value of **sindex** on a single line.

The second watch expression for **current** generates much more output. Structure members display on separate lines and are indented to reflect the nesting level. The last few lines display the data stored in the **analog** array.

The watch window updates at the end of each execution command (StepInto, StepOut, or Go). You may configure dScope to periodically update the watch window during execution by selecting the Update Watch Window command from the Setup menu.



## 6

## Breakpoints

You use breakpoints to stop program execution on a given address or a specified condition. Execution breakpoints are the simplest form; a function address or line number specifies where to stop execution.

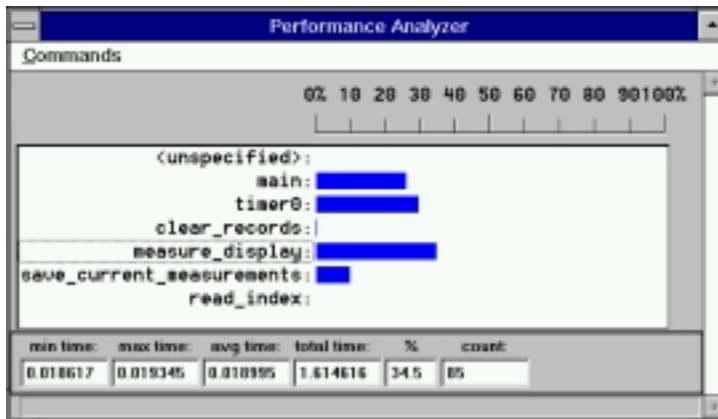


These commands create the performance analyzer address ranges for timing statistics. You may create or view the ranges with the Setup Performance Analyzer command in the Setup menu.

Perform the following steps to watch the performance analyzer in action:

1. Open the performance analyzer window using the button on the tool bar. The display shows the ranges defined above. The `<unspecified>` line accumulates all execution time outside the defined ranges,
2. Reset dScope,
3. Start program execution by clicking on the Go button in the debug window,
4. Select the serial window and type **S Enter D Enter**.

The performance analyzer window shows a bar graph for each range.



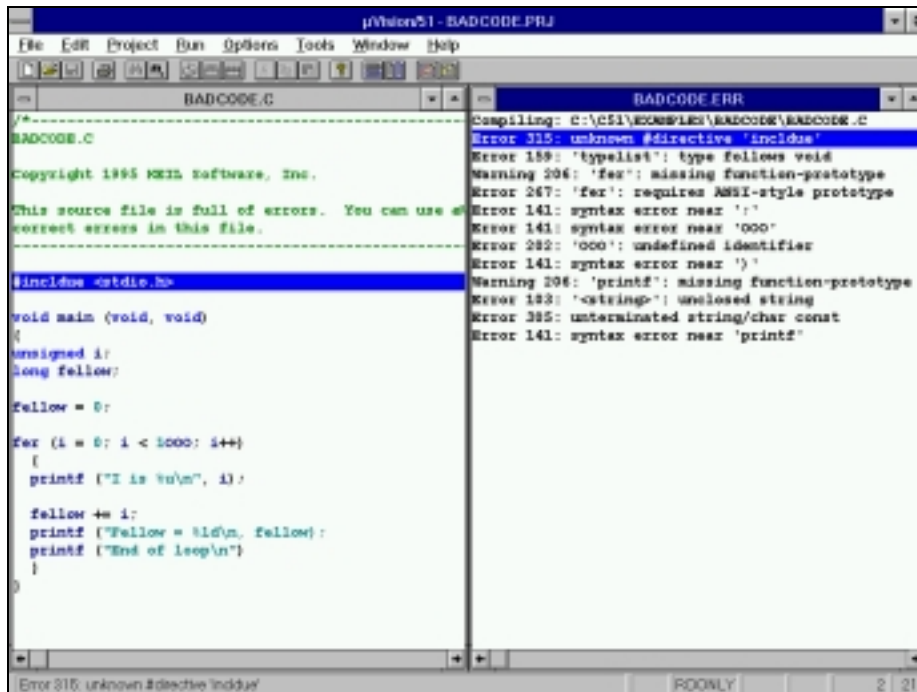
The bar graph is dynamically updated and shows the percent of the time spent executing code in each range. Click on the range to see timing statistics for each individual range.

## BADCODE: An Example with Syntax Errors

The `\C51\EXAMPLES\BADCODE\` directory contains a file called `BADCODE.C`. This file is used to demonstrate how  $\mu$ Vision interacts with the compiler to help you locate errors and warnings in your source program.

Open the `BADCODE.C` file using the Open command in the File menu. Select the Compile File command from the Project menu to compile the file. After compilation,  $\mu$ Vision determines that there are errors and displays an error window for you to peruse.

You may use the cursor keys in the error window to scroll through the errors generated by the compiler. As you move from line to line, the source window is updated to reflect the line on which the error was encountered.



The screenshot shows the  $\mu$ Vision IDE with two windows open. The left window, titled `BADCODE.C`, displays the source code for `BADCODE.C`. The right window, titled `BADCODE.ERR`, displays the compilation errors and warnings. The error window is currently showing the first error, `Error 315: unknown #directive 'include'`.

```

BADCODE.C
-----
Copyright 1995 KEEL Software, Inc.

This source file is full of errors. You can use a
correct errors in this file.
-----
#include <stdio.h>

void main (void, void)
{
  unsigned i;
  long fellow;

  fellow = 0;

  for (i = 0; i < 1000; i++)
  {
    printf ("I is %u\n", i);
    fellow += i;
    printf ("Fellow = %ld\n, fellow);
    printf ("End of loop\n");
  }
}

BADCODE.ERR
-----
Compiling: C:\C51\EXAMPLES\BADCODE\BADCODE.C
Error 315: unknown #directive 'include'
Error 189: 'typelist': type follows void
Warning 206: 'fer': missing function-prototype
Error 247: 'fer': requires ANSI-style prototype
Error 141: syntax error near ';'
Error 141: syntax error near 'ooo'
Error 282: 'ooo': undefined identifier
Error 141: syntax error near ')'
Warning 206: 'printf': missing function-prototype
Error 183: '<string>': unclosed string
Error 385: unterminated string/char const
Error 141: syntax error near 'printf'
  
```

When the error window displays, it may cover a portion of the source window. Use the tile vertical or tile horizontal button to display the windows side-by-side.





## Chapter 7. Hardware Products

Keil Software offers a number of hardware products that you can use to assist in 8051/251 software development. Currently, our hardware products include:

- ProROM EPROM Emulator,
- MCB517A Evaluation Board,
- MCB251SB Evaluation Board.

Each of these products is described in the following sections.

### ProROM EPROM Emulator

ProROM is an EPROM emulator that connects between the parallel printer port of your PC and the ROM socket of your target hardware. With ProROM, you can rapidly develop and test your embedded target program.

It only takes a few seconds to download 64 Kbytes of program code to ProROM. You no longer have to rely on or wait for EPROM programmers and erasers that may take several minutes between software iterations.

ProROM comes with an easy to use loader program that downloads your binary or Intel HEX files. Additionally, you can use ProROM with the  $\mu$ Vision development environment to automate your build and load development cycle.

The ProROM EPROM emulator comes complete with:

- User's Manual,
- Software and file conversion utilities,
- ProROM EPROM Emulator,
- 28-pin DIP interface cable,
- PC parallel-port cable.

ProROM provides a quick, convenient solution for rapid software development.

## MCB517A Evaluation Board

The MCB517A evaluation board is a single board computer that supports the Siemens 80C517(A) microcontroller. The MCB517A lets you write and test code for the 80C517(A) using the Keil Software 8051 development tools and the 8051 monitor.

The MCB517A includes a user's manual that clearly describes the board and an evaluation kit that includes a 2 Kbyte size-limited tool set. The tools provided include:

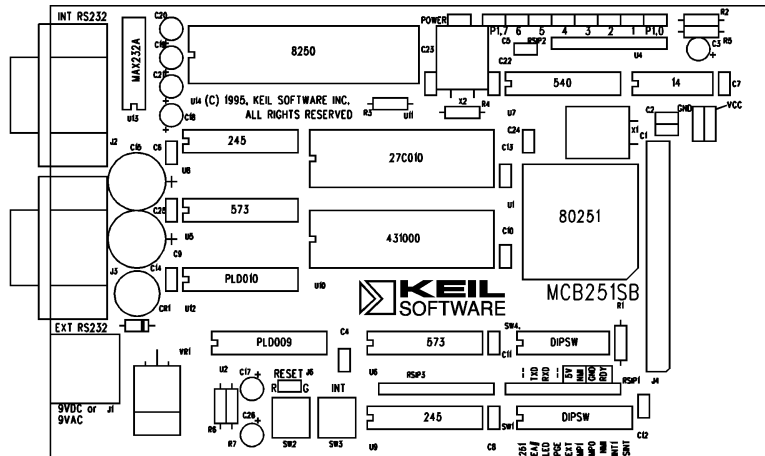
- The C51 compiler,
- A51 assembler,
- $\mu$ Vision/51 IDE for Windows,
- dScope-51 simulator for Windows,
- 8051 Monitor program and dScope interface DLLs,
- all the necessary utilities,
- and several example programs.

The 8051 monitor lets you download and execute 8051 applications you develop using the tools included with the package. You can build applications using  $\mu$ Vision and the C51 compiler and A51 assembler, and you can test and debug applications using dScope and the monitor.

The MCB517A is a complete starter package for anyone interested in the Siemens 517. Since the Siemens 517 CPU is a superset of the 8051 and 80515 the MCB517A board can be used also for projects using the 8051, 80C515(A) and 80C517(A). The MCB517A uses for communication with the Monitor the 2nd serial interface of the 517 CPU, this frees up the standard 8051 serial interface for the user application. The MCB517A is a complete starter package for anyone interested in the Siemens 517.

## MCB251SB Evaluation Board

The MCB251SB evaluation board is a single board computer that supports the Intel 80C251SB microcontroller. The MCB251SB lets you evaluate all operating modes of the 251 including page mode, non-page mode, source mode, and binary mode. Board configuration is accomplished using clearly labeled DIP switches.



The MCB251SB includes a user's manual that describes the board and data books that describe the 251 architecture. A 2 Kbyte size limited tool set is also included with the MCB251SB. The tools provided include:

- The C251 compiler,
- A251 assembler,
- $\mu$ Vision/251 IDE for Windows,
- dScope-251 simulator for Windows,
- 251 Monitor program and dScope interface DLLs,
- all the necessary utilities and example programs to help you get started.

The 251 monitor program comes installed on the board. The monitor lets you download and execute 251 applications you develop using the tools included with the package. You can build applications using  $\mu$ Vision and the C251 compiler and A251 assembler, and you can test and debug applications using dScope and the monitor.

The MCB251SB is a complete starter package for anyone interested in the Intel 251.

## Chapter 8. Real-Time Kernels

This chapter discusses the different real-time operating systems that are available for the 8051 and 251 microcontrollers.

### RTX-51 Real-Time Operating System

The RTX-51 real-time operating system is a multitasking kernel for the 8051 family of processors that simplifies the software design of complex, time-critical applications.

There are two distinct versions of RTX-51:

**RTX-51 Full** which performs both round-robin and preemptive task switching using up to four task priorities. RTX-51 Full works in parallel with interrupt functions. Signals and messages may be passed between tasks using a mailbox system. You can allocate and free memory from a memory pool. You can force a task to wait for an interrupt, time-out, or signal or message from another task or interrupt.

**RTX-51 Tiny** which is a subset of RTX-51 Full. RTX-51 Tiny easily runs on single-chip 8051 systems without any external data memory. RTX-51 Tiny supports many of the features found in RTX-51 Full with the following exceptions:

1. Task switching is accomplished by round-robin multitasking and signals.
2. Preemptive task switching is not supported.
3. No message routines are included.
4. No memory pool allocation routines are available.

The rest of this section uses RTX-51 to refer to RTX-51 Full and RTX-51 Tiny. Differences between the two are stated where applicable.

### Introduction

Many microcontroller applications require simultaneous execution of multiple jobs or tasks. For such applications, a real-time operating system (RTOS) allows

flexible scheduling of system resources (CPU, memory, etc.) to several tasks. RTX-51 implements a powerful RTOS which is easy to use. RTX-51 works with all 8051 derivatives.

You write and compile RTX-51 programs using standard C constructs and compiling them with C51. Only a few deviations from standard C are required in order to specify the task ID and priority. RTX-51 programs also require that you include the real-time executive header file and link using the BL51 code banking linker/locator and the appropriate RTX-51 library file.

## Single Task Program

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

```
int counter;

void main (void) {
    counter = 0;

    while (1) {
        counter++;
    }
}
```

/\* repeat forever \*/  
/\* increment counter \*/

## Round-Robin Program

A more sophisticated C program may implement what is called a round-robin pseudo-multitasking scheme without using a RTOS. In this scheme, tasks or functions are called iteratively from within an endless loop. For example:

```
int counter;

void main (void) {
    counter = 0;

    while (1) {
        check_serial_io ();
        process_serial_cmds ();

        check_kbd_io ();
        process_kbd_cmds ();

        adjust_ctrlr_parms ();

        counter++;
    }
}
```

/\* repeat forever \*/  
/\* process serial input \*/  
/\* process keyboard input \*/  
/\* adjust the controller \*/  
/\* increment counter \*/

```
}  
}
```

## Round-Robin Scheduling With RTX-51

RTX-51 also performs round-robin multitasking which allows quasi-parallel execution of several endless loops or tasks. Tasks are not executed concurrently but are time-sliced. The available CPU time is divided into time slices and RTX-51 assigns a time slice to every task. Each task is allowed to execute for a predetermined amount of time. Then, RTX-51 switches to another task that is ready to run and allows that task to execute for a while. The time slices are very short, usually only a few milliseconds. For this reason, it appears as though the tasks are executing simultaneously.

RTX-51 uses a timing routine which is interrupt driven by one of the 8051 hardware timers. The periodic interrupt that is generated is used to drive the RTX-51 clock.

RTX-51 does not require you to have a main function in your program. It automatically begins executing task 0. If you do have a main function, you must manually start RTX-51 using the `os_create_task` function in RTX-51 Tiny and the `os_start_system` function in RTX-51.

The following example shows a simple RTX-51 application that uses only round-robin task scheduling. The two tasks in this program are simple counter loops. RTX-51 starts executing task 0 which is the function names `job0`. This function adds another task called `job1`. After `job0` executes for a while, RTX-51 switches to `job1`. After `job1` executes for a while, RTX-51 switches back to `job0`. This process is repeated indefinitely.

```
#include <rtx51tiny.h>  
  
int counter0;  
int counter1;  
  
void job0 (void) _task_ 0 {  
    os_create_task (1);                /* mark task 1 as ready */  
    while (1) {                        /* loop forever */  
        counter0++;                    /* update the counter */  
    }  
}  
  
void job1 (void) _task_ 1 {  
    while (1) {                        /* loop forever */  
        counter1++;                    /* update the counter */  
    }  
}
```

## RTX-51 Events

Rather than waiting for a task's time slice to be up, you can use the `os_wait` function to signal RTX-51 that it can let another task begin execution. This function suspends execution of the current task and waits for a specified event to occur. During this time, any number of other tasks may be executing.

### Using Time-outs with RTX-51

The simplest event you can wait for with the `os_wait` function is a time-out period in RTX-51 clock ticks. This type of event can be used in a task where a delay is required. This could be used in code that polled a switch. In such a situation, the switch need only be checked every 50ms or so.

The next example shows how you can use the `os_wait` function to delay execution while allowing other tasks to execute.

```
#include <rtx51tny.h>

int counter0;
int counter1;

void job0 (void) _task_ 0 {
    os_create_task (1);                /* mark task 1 as ready */
    while (1) {                        /* loop forever */
        counter0++;                    /* update the counter */
        os_wait (K_TMO, 3, 0);        /* pause for 3 clock ticks */
    }
}

void job1 (void) _task_ 1 {
    while (1) {                        /* loop forever */
        counter1++;                    /* update the counter */
        os_wait (K_TMO, 5, 0);        /* pause for 5 clock ticks */
    }
}
```

# 8

In the above example, `job0` enables `job1` as before. But now, after incrementing `counter0`, `job0` calls the `os_wait` function to pause for 3 clock ticks. At this time, RTX-51 switches to the next task, which is `job1`. After `job1` increments `counter1`, it too calls `os_wait` to pause for 5 clock ticks. Now, RTX-51 has no other tasks to execute, so it enters an idle loop waiting for 3 clock ticks to elapse before it can continue executing `job0`.

The result of this example is that `counter0` gets incremented every 3 timer ticks and `counter1` gets incremented every 5 timer ticks.



## Using Signals with RTX-51

You can use the `os_wait` function to pause a task while waiting for a signal (or binary semaphore) from another task. This can be used for coordinating two or more tasks. Waiting for a signal works as follows: If a task goes to wait for a signal, and the signal flag is 0, the task is suspended until the signal is sent. If the signal flag is already 1 when the task queries the signal, the flag is cleared, and execution of the task continues. The following example illustrates this:

```
#include <rtx51tny.h>

int counter0;
int counter1;

void job0 (void) _task_ 0 {
    os_create_task (1);                               /* mark task 1 as ready */
    while (1) {                                       /* loop forever */
        if (++counter0 == 0)                          /* update the counter */
            os_send_signal (1);                       /* signal task 1 */
    }
}

void job1 (void) _task_ 1 {
    while (1) {                                       /* loop forever */
        os_wait (K_SIG, 0, 0);                        /* wait for a signal */
        counter1++;                                   /* update the counter */
    }
}
```

In the above example, `job1` waits until it receives a signal from any other task. When it does receive a signal, it increments `counter1` and again waits for another signal. `job0` continuously increments `counter0` until it overflows to 0. When that happens, `job0` sends a signal to `job1` and RTX-51 marks `job1` as ready for execution. `job1` does not start until RTX-51 gets its next timer tick.

## Priorities and Preemption

One disadvantage of the above program example is that `job1` is not started immediately when it is signaled by `job0`. In some circumstances, this is unacceptable for timing reasons. RTX-51 allows you to assign priority levels to tasks. When a higher priority task becomes available, it interrupts or preempts a lower priority task. This is called preemptive multitasking or just preemption.

---

### NOTE

*Preemption and priority levels are not supported by RTX-51 Tiny.*

---

You can modify the above function declaration for `job1` to give it a higher priority than `job0`. By default, all tasks are assigned a priority level of 0. This is the lowest priority level. The priority level can be 0 through 3. The following example shows how to define `job1` with a priority level of 1.

```
void job1 (void) _task_ 1 _priority_ 1 {
    while (1) {
        os_wait (K_SIG, 0, 0);
        counter1++;
    }
}
```

Now, whenever `job0` sends a signal to `job1`, `job1` starts immediately.

## Compiling and Linking with RTX-51

RTX-51 is fully integrated into the C51 programming language. This makes generating RTX-51 applications very easy to master. You do not need to write any 8051 assembly routines or functions. You only have to compile your RTX-51 programs with C51 and link them with the BL51 code banking linker/locator.

For example, you should use the following command lines with RTX-51 Tiny.

```
C51 EXAMPLE.C
BL51 EXAMPLE.OBJ RTX51TINY
```

Use the following command lines to compile and link with RTX-51.

```
C51 EXAMPLE.C
BL51 EXAMPLE.OBJ RTX51
```

## Interrupts

RTX-51 works in parallel with interrupt functions. Interrupt functions can communicate with RTX-51 and can send signals or messages to RTX-51 tasks. RTX-51 Full lets you assign an interrupt to a task.

## Message Passing

RTX-51 Full supports message exchange between tasks with the following functions: `isr_recv_message`, `isr_send_message`, `os_send_message`, and `os_wait`.

A message is a 16-bit value which can be interpreted as a number or as a pointer to a memory block. RTX-51 Full supports variable sized messages using a memory pool system.

## CAN Communication

Controller Area Networks are easily implemented with RTX-51/CAN. RTX-51/CAN is a CAN task integrated into RTX-51 Full. An RTX-51 CAN task implements message passing via the CAN network. Other CAN stations can be configured either with or without RTX-51.

## BITBUS Communication

RTX-51 Full includes both master and slave BITBUS tasks supporting message passing with the Intel 8044.

## Events

RTX-51 supports the following events for the `os_wait` function:

- A **Timeout** suspends the running task for a defined number of clock ticks.
- An **Interval** is similar to a **timeout**, however, the **interval** is intended for use with tasks that must execute synchronously.
- **Signals** are used for inter-task coordination.
- **Messages** are used for exchange of messages. †
- An **Interrupt** lets a task wait for an 8051 hardware interrupt. †
- **Semaphores** are used for management of shared system resources. †

† These events are available only in RTX-51 Full.

## RTX-51 Functions

The following table lists some of the RTX-51 functions along with a brief description and execution timing (for RTX-51 Full).

Function	Description	CPU Cycles
<b>isr_rcv_message</b> †	Receive a message (call from interrupt).	71 (with message)
<b>isr_send_message</b> †	Send a message (call from interrupt).	53
<b>isr_send_signal</b>	Send a signal to a task (call from interrupt).	46
<b>os_attach_interrupt</b> †	Assign task to interrupt source.	119
<b>os_clear_signal</b>	Delete a previously sent signal.	57
<b>os_create_task</b>	Move a task to execution queue.	302
<b>os_create_pool</b> †	Define a memory pool.	644 (size 20 * 10 bytes)
<b>os_delete_task</b>	Remove a task from execution queue.	172
<b>os_detach_interrupt</b> †	Remove interrupt assignment.	96
<b>os_disable_isr</b> †	Disable 8051 hardware interrupts.	81
<b>os_enable_isr</b> †	Enable 8051 hardware interrupts.	80
<b>os_free_block</b> †	Return a block to a memory pool.	160
<b>os_get_block</b> †	Get a block from a memory pool.	148
<b>os_send_message</b> †	Send a message (call from task).	443 with task switch
<b>os_send_signal</b>	Send a signal to a task (call from tasks).	408 with task switch 316 with fast task switch 71 without task switch
<b>os_send_token</b> †	Set a semaphore (call from task).	343 with fast task switch 94 without task switch
<b>os_set_slice</b> †	Set the RTX-51 system clock time slice.	67
<b>os_wait</b>	Wait for an event.	68 for pending signal 160 for pending message

† These functions are available only in RTX-51 Full.

Additional debug and support functions in RTX-51 Full include the following:

Function	Description
<b>oi_reset_int_mask</b>	Disables interrupt sources external to RTX-51.
<b>oi_set_int_mask</b>	Enables interrupt sources external to RTX-51.
<b>os_check_mailbox</b>	Returns information about the state of a specific mailbox.
<b>os_check_mailboxes</b>	Returns information about the state of all mailboxes in the system.
<b>os_check_pool</b>	Returns information about the blocks in a memory pool.
<b>os_check_semaphore</b>	Returns information about the state of a specific semaphore.
<b>os_check_semaphores</b>	Returns information about the state of all semaphores in the system.
<b>os_check_task</b>	Returns information about a specific task.
<b>os_check_tasks</b>	Returns information about all tasks in the system.

## CAN Functions

The CAN functions are available only with RTX-51 Full. CAN controllers supported include the Philips 82C200 and 80C592 and the Intel 82526. More CAN controllers are in preparation.

CAN Function	Description
<b>can_bind_obj</b>	Bind an object to a task; task is started when object is received.
<b>can_def_obj</b>	Define communication objects.
<b>can_get_status</b>	Get CAN controller status.
<b>can_hw_init</b>	Initialize CAN controller hardware.
<b>can_read</b>	Directly read an object's data.
<b>can_receive</b>	Receive all unbound objects.
<b>can_request</b>	Send a remote frame for the specified object.
<b>can_send</b>	Send an object over the CAN bus.
<b>can_start</b>	Start CAN communications.
<b>can_stop</b>	Stop CAN communications.
<b>can_task_create</b>	Create the CAN communication task.
<b>can_unbind_obj</b>	Disconnect the binding between a task and an object.
<b>can_wait</b>	Wait for reception of a bound object.
<b>can_write</b>	Write new data to an object without sending it.

## Technical Data

Description	RTX-51 Full	RTX-51 Tiny
Number of tasks	256; max. 19 tasks active	16
RAM requirements	40 .. 46 bytes DATA 20 .. 200 bytes IDATA (user stack) min. 650 bytes XDATA	7 bytes DATA 3 * <task count> IDATA
Code requirements	6KB .. 8KB	900 bytes
Hardware requirements	timer 0 or timer 1	timer 0
System clock	1000 .. 40000 cycles	1000 .. 65535 cycles
Interrupt latency	< 50 cycles	< 20 cycles
Context switch time	70 .. 100 cycles (fast task) 180 .. 700 cycles (standard task) depends on stack load	100 .. 700 cycles depends on stack load
Mailbox system	8 mailboxes with 8 integer entries each	not available
Memory pool system	up to 16 memory pools	not available
Semaphores	8 * 1 bit	not available

## Chapter 9. Command Reference

This chapter briefly describes the commands and controls for the Keil Software 8051 and 251 development tools. Commands and controls are listed in a tabular format along with a description. Underlined characters represent abbreviations for the particular control or directive.

### A51/A251 Macro Assemblers

#### Invocation:

```
A51 sourcefile [directives]
A251 sourcefile [directives]
A51 @commandfile
A251 @commandfile
```

where

**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**. You may use a command file to make assembling a source file easier or when you have more directives than fit on the command line.

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

A51 / A251 Controls	Meaning
<u>C</u> ASE ‡	Enables case sensitive symbol names.
<u>D</u> ATE( <i>date</i> )	Places <b>date</b> string in header (9 characters maximum).
<u>D</u> EBUG	Includes debugging symbol information in the object file.
<u>E</u> RRORPRINT[( <i>filename</i> )]	Outputs error messages to <b>filename</b> .
<u>I</u> NCLUDE( <i>filename</i> )	Includes the contents of <b>filename</b> in the assembly.
<u>M</u> ACRO	Enables standard macro processing.
<u>M</u> OD <u>B</u> IN ‡	Selects 251 binary mode (default).
<u>M</u> OD <u>S</u> R <u>C</u> ‡	Selects 251 source mode.
<u>M</u> PL	Enables Intel-style macro processing.
<u>N</u> O <u>A</u> MAKE	Excludes AutoMAKE information from the object file.
<u>N</u> O <u>C</u> OND	Excludes unassembled conditional assembly code from the listing file.
<u>N</u> O <u>G</u> EN	Disables macro expansions in the listing file.
<u>N</u> O <u>L</u> INES	Excludes line number information from the object file.

A51 / A251 Controls	Meaning
<b><u>NOLIST</u></b>	Excludes the assembler source code from the listing file.
<b><u>NOMACRO</u></b>	Disables standard macro processing.
<b><u>NOMOD251</u></b> †	Disables enhanced 251 instruction set.
<b><u>NOMOD51</u></b> †	Disables predefined 8051-specific special function registers.
<b><u>NOSYMBOLS</u></b>	Excludes the symbol table from the listing file.
<b><u>NOSYMLIST</u></b>	Excludes symbol definitions from the listing file.
<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 <b><i>filename</i></b> if specified.
<b><u>PAGELength</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>PRINT</u></b> [( <i>filename</i> )], <b><u>NOPRINT</u></b>	Enables or disables listing file output. The listing file is saved as <b><i>filename</i></b> if specified.
<b><u>REGISTERBANK</u></b> ( <i>num</i> , ...), <b><u>NOREGISTERBANK</u></b>	Indicates that one or more registerbanks are used or indicates that no register banks are used.
<b>RESET</b> ( <i>symbol</i> , ...)	Assigns a value of 0000h to the specified symbols.
<b>SET</b> ( <i>symbol</i> , ...)	Assigns a value of 0FFFFh to the specified symbols.
<b><u>TITLE</u></b> ( <i>title</i> )	Includes <b><i>title</i></b> in the listing file header.
<b><u>XREF</u></b>	Includes a symbol cross reference listing in the listing file.

† These controls are available only in the A51 macro assembler.

‡ These controls are available only in A251 macro assembler.

## C51/C251 Compiler

### Invocation:

```
C51 sourcefile [directives]
C251 sourcefile [directives]
C51 @commandfile
C251 @commandfile
```

where

**sourcefile** is the name of a C source file.

**commandfile** is the name of a file which contains a complete command line for the compiler including a **sourcefile** and **directives**. You may use a command file to make compiling a source file easier or when you have more directives than fit on the command line.

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



C51 / C251 Controls	Meaning
<u>C</u> ODE	Includes an assembly listing in the listing file.
<u>C</u> OMPACT	Selects the COMPACT memory model.
<u>D</u> EBUG	Includes debugging information in the object file.
<u>D</u> EFINE	Defines preprocessor names on the command line.
<u>F</u> LOATFUZZY	Specifies the number of bits rounded during floating-point comparisons.
<u>H</u> OLD( <i>d,n,x</i> ) ‡	Specifies size limits for variables placed in <b>data</b> ( <i>d</i> ), <b>near</b> ( <i>n</i> ), and <b>xdata</b> ( <i>x</i> ) memory areas.
<u>I</u> NTERVAL †	Specifies the interval for interrupt vectors.
<u>I</u> NTR2 ‡	Saves upper program counter byte and PSW1 in interrupt functions.
<u>I</u> NTVECTOR( <i>n</i> ), <u>N</u> OINTVECTOR	Specifies offset for interrupt table, using <i>n</i> , or excludes interrupt vectors from the object file.
<u>L</u> ARGE	Selects the LARGE memory model.
<u>L</u> ISTINGINCLUDE	Includes the contents of include files in the listing file.
<u>M</u> AXARGS( <i>n</i> )	Specifies the number of bytes reserved for variable length argument lists.
<u>M</u> OD517 †	Enables support for the additional hardware of the Siemens 80C517 and its derivatives.
<u>M</u> ODBIN ‡	Generates 251 binary mode code.
<u>M</u> ODDP2 †	Enables support for the additional hardware of Dallas Semiconductor 80C320/520/530 and the AMD 80C521.
<u>M</u> ODSRC ‡	Generates 251 source mode code.
<u>N</u> OAMAKE	Excludes AutoMAKE information from the object file.
<u>N</u> OAREGS †	Disables absolute register addressing using <b>ARn</b> instructions.
<u>N</u> OCOND	Excludes skipped conditional code from the listing file.
<u>N</u> OEXTEND	Disables 8051/251 extensions and processes only ANSI C constructs.
<u>N</u> OINTPROMOTE †	Disables ANSI integer promotion rules.
<u>N</u> OREGPARMS †	Disables passing parameters in registers.
<u>O</u> BJECT[( <i>filename</i> )], <u>N</u> OOBJECT	Enables or disables object file output. The object file is saved as <b>filename</b> if specified.
<u>O</u> BJECTEXTEND †	Includes additional variable type information in the object file.
<u>O</u> PTIMIZE	Specifies the level of optimization performed by the compiler.
<u>O</u> RDER	Locates variables in memory in the same order in which they are declared in the source file.
<u>P</u> AGELENGTH( <i>n</i> )	Sets maximum number of lines in each page of listing file.
<u>P</u> AGewidth( <i>n</i> )	Sets maximum number of characters in each line of listing file.
<u>P</u> ARM51 ‡	Uses parameter passing conventions of the C51 compiler.

C51 / C251 Controls	Meaning
<b><u>P</u>REPRINT</b> [( <i>filename</i> )]	Produces a preprocessor listing file with all macros expanded. The preprocessor listing file is saved as <b><i>filename</i></b> if specified.
<b><u>P</u>RI</b> NT[( <i>filename</i> )], <b><u>N</u>OPRI</b> NT	Enables or disables listing file output. The listing file is saved as <b><i>filename</i></b> if specified.
<b><u>R</u>EGFILE</b> ( <i>filename</i> )	Specifies the name of the generated file to contain register usage information.
<b><u>R</u>EGISTERBANK</b> †	Selects the register bank to use functions in the source file.
<b><u>R</u>OM</b> ({ <b><u>S</u></b> SMALL  <b><u>C</u></b> OMPACT  <b><u>L</u></b> ARGE})	Controls generation of <b>AJMP</b> and <b>ACALL</b> instructions.
<b><u>S</u></b> SMALL	Selects the SMALL memory model.
<b><u>S</u></b> RC	Creates an assembly source file instead of an object file.
<b><u>S</u></b> YMBOLS	Includes a list of the symbols used in the listing file.
<b><u>W</u></b> ARNINGLEVEL( <i>n</i> )	Controls the types and severity of warnings generated.

† These controls are available only in the C51 compiler.

‡ These controls are available only in C251 compiler.

## L51/BL51 Linker/Locator

### Invocation:

```
BL51 inputlist [TO outputfile] [directives]
L51 inputlist [TO outputfile] [directives]
BL51 @commandfile
L51 @commandfile
```

where

**inputlist** is a list of the object files and libraries, separated by commas, that the linker includes in the final 8051 application.

**outputfile** is the name of the absolute object module the linker creates.

**commandfile** is the name of a file which contains a complete command line for the linker/locator including an **inputlist** and **directives**. You may use a command file to make linking your application easier or when you have more input files or more directives than fit on the command line.

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

BL51 Controls	Meaning
<b><u>B</u></b> ANKAREA ‡	Specifies the address range where the code banks are located.

BL51 Controls	Meaning
<b><u>BANK</u><sub>x</sub> ‡</b>	Specifies the starting address, segments, and object modules for code banks 0 to 31.
<b><u>BIT</u></b>	Locates and orders <b>BIT</b> segments.
<b><u>CODE</u></b>	Locates and orders <b>CODE</b> segments.
<b><u>COMMON</u> ‡</b>	Specifies the starting address, segments, and object modules to place in the common bank. This directive is essentially the same as the <b>CODE</b> directive.
<b><u>DATA</u></b>	Locates and orders <b>DATA</b> segments.
<b><u>IDATA</u></b>	Locates and orders <b>IDATA</b> segments.
<b><u>IXREF</u></b>	Includes a cross reference report in the listing file.
<b><u>NAME</u></b>	Specifies a module name for the object file.
<b><u>NOAMAKE</u></b>	Excludes AutoMAKE information from the object file.
<b><u>NODEBUGLINES</u></b>	Excludes line number information from the object file.
<b><u>NODEBUGPUBLICS</u></b>	Excludes public symbol information from the object file.
<b><u>NODEBUGSYMBOLS</u></b>	Excludes local symbol information from the object file.
<b><u>NODEFAULTLIBRARY</u></b>	Excludes modules from the run-time libraries.
<b><u>NOLINES</u></b>	Excludes line number information from the listing file.
<b><u>NOMAP</u></b>	Excludes memory map information from the listing file.
<b><u>NOOVERLAY</u></b>	Prevents overlaying or overlapping local <b>BIT</b> and <b>DATA</b> segments.
<b><u>NOPUBLICS</u></b>	Excludes public symbol information from the listing file.
<b><u>NOSYMBOLS</u></b>	Excludes local symbol information from the listing file.
<b><u>OVERLAY</u></b>	Directs the linker to overlay local data & bit segments and lets you change references between segments.
<b><u>PAGELLENGTH</u>(n)</b>	Sets maximum number of lines in each page of listing file.
<b><u>PAGEWIDTH</u>(n)</b>	Sets maximum number of characters in each line of listing file.
<b><u>PDATA</u></b>	Specifies the starting address for <b>PDATA</b> segments.
<b><u>PRECEDE</u></b>	Locates and orders segments that should precede all others in the internal data memory.
<b><u>PRINT</u></b>	Specifies the name of the listing file.
<b><u>RAMSIZE</u></b>	Specifies the size of the on-chip data memory.
<b><u>REGFILE</u>(filename)</b>	Specifies the name of the generated file to contain register usage information.
<b><u>RTX51</u> ‡</b>	Includes support for the RTX-51 full real-time kernel.
<b><u>RTX51TINY</u> ‡</b>	Includes support for the RTX-51 tiny real-time kernel.
<b><u>STACK</u></b>	Locates and orders <b>STACK</b> segments.
<b><u>XDATA</u></b>	Locates and orders <b>XDATA</b> segments.

‡ These controls are available only in the BL51 code banking linker/locator.

## L251 Linker/Locator

### Invocation:

```
L251 inputlist [TO outputfile] [directives]
L251 @commandfile
```

where

- inputlist** is a list of the object files and libraries, separated by commas, that the linker includes in the final 251 application.
- outputfile** is the name of the absolute object module the linker creates.
- commandfile** is the name of a file which contains a complete command line for the linker/locator including an **inputlist** and **directives**. You may use a command file to make linking your application easier or when you have more input files or more directives than fit on the command line.
- directives** are control parameters which are described in the following table.

L251 Controls	Meaning
<b><u>A</u>SSIGN</b>	Defines public symbols on the command line.
<b><u>C</u>LASSES</b>	Specifies a physical address range for segments in a memory class.
<b><u>I</u>XREF</b>	Includes a cross reference report in the listing file.
<b><u>N</u>AME</b>	Specifies a module name for the object file.
<b><u>N</u>OAMAKE</b>	Excludes AutoMAKE information from the object file.
<b><u>N</u>O<u>C</u>OMMENTS</b>	Excludes comment information from the listing file and the object file.
<b><u>N</u>ODEFAULT<u>L</u>IBRARY</b>	Excludes modules from the run-time libraries.
<b><u>N</u>OLINES</b>	Excludes line number information from the listing file and object file.
<b><u>N</u>OMAP</b>	Excludes memory map information from the listing file.
<b><u>N</u>O<u>O</u>VERLAY</b>	Prevents overlaying or overlapping local <b>BIT</b> and <b>DATA</b> segments.
<b><u>N</u>OPUBLICS</b>	Excludes public symbol information from the listing file and the object file.
<b><u>N</u>OSYMBOLS</b>	Excludes local symbol information from the listing file.
<b><u>N</u>OTYPES</b>	Excludes type information from the listing file and the object file.
<b><u>O</u>BJECT<u>C</u>ONTROLS</b>	Excludes specific debugging information from the object file. Subcontrols must be specified in parentheses. See <b>NOCOMMENTS</b> , <b>NOLINES</b> , <b>NOPUBLICS</b> , <b>NOSYMBOLS</b> , and <b>PURGE</b> .

L251 Controls	Meaning
<b><u>O</u>VERLAY</b>	Directs the linker to overlay local data & bit segments and lets you change references between segments.
<b><u>P</u>AGEL<u>E</u>NGTH(<i>n</i>)</b>	Sets maximum number of lines in each page of listing file.
<b><u>P</u>AGE<u>W</u>IDTH(<i>n</i>)</b>	Sets maximum number of characters in each line of listing file.
<b><u>P</u>RI<u>N</u>T</b>	Specifies the name of the listing file.
<b><u>P</u>RI<u>N</u>T<u>C</u>ON<u>T</u>RO<u>L</u>S</b>	Excludes specific debugging information from the listing file. Subcontrols must be specified in parentheses. See <b>NOCOMMENTS</b> , <b>NOLINES</b> , <b>NOPUBLICS</b> , <b>NOSYMBOLS</b> , and <b>PURGE</b> .
<b><u>P</u>UR<u>G</u>E</b>	Excludes all debugging information from the listing file and the object file.
<b><u>R</u>AM<u>S</u>I<u>Z</u>E</b>	Specifies the size of the on-chip data memory.
<b><u>R</u>EG<u>F</u>I<u>L</u>E(<i>filename</i>)</b>	Specifies the name of the generated file to contain register usage information.
<b><u>R</u>ES<u>E</u>R<u>V</u>E</b>	Reserves memory ranges and prevents the linker from using these memory areas.
<b>RTX251</b>	Includes support for the RTX-251 full real-time kernel.
<b>RTX251TINY</b>	Includes support for the RTX-251 tiny real-time kernel.
<b><u>S</u>EG<u>M</u>ENTS</b>	Defines physical memory addresses and orders for specified segments.
<b>SE<u>G</u>SI<u>Z</u>E</b>	Specifies memory space used by a segment.
<b><u>W</u>AR<u>N</u>I<u>N</u>G<u>L</u>E<u>V</u>EL(<i>n</i>)</b>	Controls the types and severity of warnings generated.

## OC51 Banked Object File Converter

**Invocation:** `OC51 banked_file`

where

`banked_file` is the name of a banked object file.

## OH51 Object-Hex Converter

**Invocation:** `OH51 absfile [HEXFILE(hexfile)]`

where

`absfile` is the name of an absolute object file.

`hexfile` is the name of the Intel HEX file to create.

## OH251 Object-Hex Converter

### Invocation:

```
OH251 absfile [HEXFILE(hexfile)] [{HEX|H386}]
[RANGE(start-end)]
```

where

- absfile** is the name of an absolute object file.
- hexfile** is the name of the HEX file to create.
- HEX** specifies that a standard Intel HEX file is created.
- H386** specifies that an Intel HEX-386 file is created.
- RANGE** specifies the address range of data in the **absfile** to convert and store in the HEX file. The default range is 0xFF0000 to 0xFFFFFFFF.
- start** specifies the starting address of the range. This address must be entered in C hexadecimal notation, for example: 0xFF0000.
- end** specifies the ending address of the range. This address must be entered in C hexadecimal notation, for example: 0xFFFFFFFF.

## LIB51/LIB251 Library Manager

### Invocation:

```
LIB51 [command]
LIB251 [command]
```

where

- command** is a control command described in the following table. If no command is given LIB51 enters an interactive command mode.

LIB51 / LIB251 Command	Meaning
<u>A</u> DD	Adds an object module to the library file.
<u>C</u> REATE	Creates a new library file.
<u>D</u> ELETE	Removes an object module from the library file.
<u>E</u> XIT	Exits the library manager interactive mode.
<u>H</u> ELP	Displays help information for the library manager.
<u>L</u> IST	Displays module and public symbol information stored in the library file.

# Index

<hr/>	
μVision	
Editor .....	59
Menu Commands .....	59
Options.....	60
Overview.....	56
Project Manager.....	60
Starting.....	56
μVision/251 for Windows .....	54
μVision/51 for Windows .....	43
251 Development Tools .....	46
8051 Development Tools .....	21
8051 Microcontroller Family.....	21
8051/251 Compiler Kit	
Subscription.....	19
8051/251 Developer's Kit	
Subscription.....	18
8051/251 Development Tools .....	21,45
8051/251 Product Line .....	11
<b>A</b>	
<hr/>	
A251 .....	17
A251 Assembler .....	51
Functional Overview.....	51
Listing File Example.....	51
A251 Macro Assembler Kit.....	17
A51 .....	15
A51 Assembler .....	37
Configuration .....	37
Functional Overview.....	37
Listing File Example.....	37
A51 Macro Assembler Kit.....	15
Additional items, document	
conventions.....	iv
alien .....	31
asm .....	31
AUTOEXEC.BAT .....	9
<b>B</b>	
<hr/>	
Backing Up Your Disks.....	6
BL51 code banking	
linker/locator .....	39
Code Banking.....	39
Common Area .....	40
Data Address Management.....	39
Executing Functions in Other	
Banks.....	40
Listing File Example .....	41
bold capital text, use of .....	iv
braces, use of.....	iv
<b>C</b>	
<hr/>	
C251 Compiler.....	46
Data Types .....	47
Listing File Example .....	49
Memory Models .....	48
Memory Selector .....	47
Program Size .....	48
Reentrant Code.....	49
Register Optimization.....	49
Run-Time Library.....	49
C251 Compiler Kit.....	16
C251 Developer's Kit .....	16
C51 Compiler.....	22
Code Optimizations.....	32
Compact model .....	26
Data Types .....	24
Debugging .....	34
Function Return Values.....	30
Generic Pointers.....	27
Interfacing to Assembly .....	30
Interfacing to PL/M-51.....	31
Interrupt Functions .....	29
Language Extensions.....	23
Large model .....	26
Library Routines.....	35
Listing File Example .....	35
Memory Models .....	26
Memory Specific Pointers .....	27
Memory Types .....	24
Parameter Passing .....	29
Pointers .....	27
Real-Time Operating System	
Support.....	30

Reentrant Functions ..... 28  
 Register Optimizing ..... 30  
 Small model ..... 26  
 C51 Compiler Kit ..... 14  
 C51 Developer's Kit ..... 14  
 C51 Professional Developer's  
 Kit ..... 13  
 CA251 ..... 16  
 CA51 ..... 14  
 can\_bind\_obj ..... 109  
 can\_def\_obj ..... 109  
 can\_get\_status ..... 109  
 can\_hw\_init ..... 109  
 can\_read ..... 109  
 can\_receive ..... 109  
 can\_request ..... 109  
 can\_send ..... 109  
 can\_start ..... 109  
 can\_stop ..... 109  
 can\_task\_create ..... 109  
 can\_unbind\_obj ..... 109  
 can\_wait ..... 109  
 can\_write ..... 109  
 Changes to the Documentation ..... 3  
 Choices, document conventions ..... iv  
 COMPACT ..... 25,26  
 CONFIG.SYS ..... 6  
 courier typeface, use of ..... iv

**D**

---

DEBUG ..... 37,51  
 Demo Kit ..... 2  
 Directory Structure ..... 7  
 Disk Cache ..... 10  
 Displayed text, document  
 conventions ..... iv  
 DK251 ..... 16  
 DK51 ..... 14  
 Document conventions ..... iv  
 Documentation Changes ..... 3  
 DOS-Based Product Installation ..... 6  
 DOS-based tool requirements ..... 5  
 double brackets, use of ..... iv  
 DS251 ..... 17  
 DS51 ..... 15  
 dScope  
 Breakpoints ..... 69

Code Coverage ..... 70  
 Command Window ..... 66  
 CPU Simulation ..... 63  
 Debug Window ..... 65  
 Functions ..... 69  
 Overview ..... 62  
 Performance Analyzer  
 Window ..... 68  
 Serial Window ..... 67  
 Starting ..... 56  
 Watch Window ..... 67  
 dScope-251 for Windows ..... 53  
 dScope-251 Simulator Kit ..... 17  
 dScope-51 for Windows ..... 42  
 dScope-51 Simulator Kit ..... 15

## E

---

ellipses, use of ..... iv  
 ellipses, vertical, use of ..... iv  
 endasm ..... 31  
 Environment Settings ..... 8  
 Evaluation Kit ..... 2  
 Evaluation Users ..... 2  
 Experienced Users ..... 3

## F

---

Filename, document conventions ..... iv  
 FR251 ..... 18  
 FR51 ..... 15

## G

---

Global Register Optimization ..... 33

## H

---

Help ..... 3  
 HOLD ..... 48

## I

---

Improving System Performance ..... 9  
 Installation ..... 5  
 Installing the Software ..... 6  
 interrupt ..... 29  
 Introduction ..... 1  
 isr\_rcv\_message ..... 106,108



isr_send_message .....	106,108
isr_send_signal .....	108
italicized text, use of.....	iv

## K

---

Key names, document conventions.....	iv
--------------------------------------	----

## L

---

L251 linker/locator .....	52
LARGE.....	25,26
LIB251 library manager .....	53
LIB51 library manager .....	42

## M

---

Manual Topics .....	1
Map files.....	41
MCB251SB Evaluation Board .....	99
MCB517A Evaluation Board .....	98
MCS <sup>®</sup> 251 Microcontroller Family.....	45

## N

---

New Users .....	2
NOMOD51 .....	37
NOOVERLAY .....	39
NOREGPparms.....	29,30

## O

---

OBJECTEXTEND .....	34
OC51 Banked Object File Converter .....	42
OH251 Object-Hex Converter.....	53
OH51 Object-Hex Converter.....	42
oi_reset_int_mask.....	108
oi_set_int_mask.....	108
OMF251 .....	46
OMF51 .....	22,31,34
Omitted text, document conventions.....	iv
Optional items, document conventions.....	iv
os_attach_interrupt.....	108
os_check_mailbox .....	108

os_check_mailboxes .....	108
os_check_pool .....	108
os_check_semaphore .....	108
os_check_semaphores.....	108
os_check_task .....	108
os_check_tasks.....	108
os_clear_signal.....	108
os_create_pool .....	108
os_create_task.....	108
os_delete_task.....	108
os_detach_interrupt.....	108
os_disable_isr .....	108
os_enable_isr .....	108
os_free_block.....	108
os_get_block.....	108
os_send_message .....	106,108
os_send_signal .....	108
os_send_token.....	108
os_set_slice .....	108
os_wait.....	106,108
OVERLAY .....	39

## P

---

PK51 .....	13
Printed text, document conventions .....	iv
ProROM EPROM Emulator .....	97

## R

---

RAM Disk.....	9
README.TXT.....	3
reentrant .....	28,29,49
REGPARMS.....	29
Reporting a problem .....	3
Requesting Assistance.....	3
ROM .....	48
RTX-251 Full Real-Time Kernel .....	18
RTX-51 .....	101
BITBUS Communication .....	107
CAN Communication.....	107
Compiling.....	106
Events.....	104,107
Functions.....	108
Interrupts .....	106
Introduction.....	101
Linking .....	106

Message Passing .....	106
Preemption .....	105
Priorities.....	105
Round-Robin Scheduling .....	103
Technical Data .....	110
Using Signals .....	105
Using Time-outs .....	104
RTX-51 Full Real-Time Kernel.....	15

## **S**

---

sans serif typeface, use of .....	iv
SCA251 .....	19
SDK251 .....	18
SMALL.....	25,26
SRC.....	31
System Requirements.....	5

## **T**

---

Technical Support.....	3
------------------------	---

Temporary Files .....	9
Types of Users.....	2

## **U**

---

User .....	2
using .....	29

## **V**

---

Variables, document conventions.....	iv
vertical bar, use of .....	iv

## **W**

---

What's Included .....	2
Windows-Based Product	
Installation.....	7
Windows-based tool	
requirements .....	5