READS166 Users Guide

Version 3.1x

February 1999

RIGEL CORPORATION
PO Box 90040
Gainesville, Florida 32607
(352) 373-4629
FAX (352) 373-1786
www.rigelcorp.com
tech@rigelcorp.com

Legal Notice:

The abbreviation PC used throughout this guide refers to the IBM Personal Computer or its compatibles. IBM PC is a trademark of International Business Machines, Inc. MS Windows is a trademark of Microsoft, Inc.

Information in this document is provided solely to enable use of Rigel products. Rigel assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Rigel products except as provide in RIGEL’s Customer Agreement for such products.

Rigel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors, which may appear in this document, nor does it make a commitment to update the information contained herein.

Rigel retains the right to make changes to these specifications at any time without notice.

Contact Rigel Corporation or your Distributor to obtain the latest specifications before placing your order.

Our Policy:

We attempt to up date all software, software manuals, and hardware manuals every 3-6 months. If there is a problem with the software or documentation it is corrected immediately. The newest version is then put on our home page (www.rigelcorp.com) with the date noted. Documentation is coded with version number and a date. The version number is the version of the board/software (V3.1), and the date (Feb1999) refers to the last date the document was written.

We welcome any and all comments about our products.

WARRANTY

RIGEL CORPORATION- CUSTOMER AGREEMENT

1. **READS 166** (referred to as simply READS) License. The READS being purchased is hereby licensed to you on a non-exclusive basis for use in only one computer system and shall remain the property of Rigel Corporation for purposes of utilization and resale. You acknowledge you may not duplicate the READS for use in additional computers, nor may you modify, disassemble, reverse engineer, translate, sub-license, rent or transfer electronically READS from one computer to another, or make it available through a timesharing service or network of computers. Rigel Corporation maintains all proprietary rights in and to READS for purposes of sale and resale or license and re-license.

   BY BREAKING THE SEAL, OPENING THE READS PACKAGE, OR LOADING IT INTO YOUR PC, YOU INDICATE YOUR ACCEPTANCE OF THIS LICENSE AGREEMENT, AS WELL AS ALL OTHER PROVISIONS CONTAINED HEREIN.

2. **Return Policy**. **This return policy applies only if you purchased the READS166 Software directly from Rigel Corporation. If purchased form a distributor please contact them for their return policy.**
   If you are not satisfied with the items purchased, prior to usage, you may return them to Rigel Corporation within thirty (30) days of your receipt of same and receive a full refund from Rigel Corporation. You will be responsible for shipping costs. Please call (352) 373-4629 prior to shipping. A refund will not be given if the READS166 Disk package has been opened.

3. **Limited Warranty**. Rigel Corporation warrants, for a period of sixty (60) days from your receipt, that READS disks, or CD ROM shall be free of substantial errors or defects in material and workmanship which will materially interfere with the proper operation of the items purchased. If you believe such an error or defect exists, please call Rigel Corporation at (352) 373-4629 to see whether such error or defect may be corrected, prior to returning items to Rigel Corporation. Rigel Corporation will repair or replace, at its sole discretion, any defective items, at no cost to you, and the foregoing shall constitute your sole and exclusive remedy in the event of any defects in material or workmanship.

   THE LIMITED WARRANTIES SET FORTH HEREIN ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

   YOU ASSUME ALL RISKS AND LIABILITY FROM OPERATION OF ITEMS PURCHASED AND RIGEL CORPORATION SHALL IN NO EVENT BE LIABLE FOR DAMAGES CAUSED BY USE OR PERFORMANCE, FOR LOSS PROFITS, PERSONAL INJURY OR FOR ANY OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES. RIGEL CORPORATION'S LIABILITY SHALL NOT EXCEED THE COST OF REPAIR OR REPLACEMENT OF DEFECTIVE ITEMS.

   IF THE FOREGOING LIMITATIONS ON LIABILITY ARE UNACCEPTABLE TO YOU, YOU SHOULD RETURN ALL ITEMS PURCHASED TO YOUR SUPPLIER.

4. **Governing Law**. This agreement and all rights of the respective parties shall be governed by the laws of the State of Florida.

Table of Contents

1.0 INTRODUCTION

READS166, version 3.1x, is Rigel Corporation's Integrated Development Environment for the C166/ST10 16-bit processors. READS166 includes an editor, a host-to-board communications system, an assembler, and a C compiler. READS166 is completely rewritten in native 32-bit code to run on Windows95, Windows98, and WindowsNT. READS166 includes a sophisticated project management system to simplify code reusability and version control. The C compiler is rewritten to support a full debugger. The debugger allows you to step through your code with breakpoints and variable watches as the compiled code runs on the target board, similar to the operation of an in-circuit emulator.

RMON166 - The READS166 monitor program

RMON 166 is downloaded after bootstrapping (or it may be placed into ROM) and supports basic memory and port functions. RMON166 allows downloading and running applications programs. The complete source code for user modifications or upgrades is included on disk.

Ra66 - The READS166 Assembler

Ra66 is an assembler for the C166 family of controllers. It is a multi-pass absolute assembler that generates HEX code directly from assembly source code.

Rc66 - The READS166 C Compiler

Rc66 is a C Compiler for the C166 family of processors. It compiles code for the tiny memory model, which fully resides in the first segment of memory. Rc66 is designed as a low-cost C compiler which provides a quick development cycle for simpler applications which do not need more than 64K of code, or the use of standard C libraries. Rc66 implements a subset of ANSI C. Rc66 works in conjunction with Ra66: first an assembly language program is generated from the C source then a HEX file is created.

The READS166 software has the following distinctive features:

- Project management for organized software development
- Archive storage for source code modules
- Multiple project management with drag and drop module transfers
- Enhanced graphical user interface for easy monitoring
- Stand alone compiler and editor applications connected to READS166 in a client/server fashion
- Drag and drop code development
- Mixed mode projects (C and assembler)
- Code wizard generators for C and Assembly
- ST10F262/ST10F272 Multiply-Accumulate (MAC) instructions

2.0 SOFTWARE SETUP

2.1 System Requirements

READS166 is designed to work with an IBM PC or compatible, 486 or better, running Windows 95, Windows 98 or Windows NT. READS166 supports the bootstrap loader feature and downloads a minimal monitor during bootstrapping. The source code and description of the bootstrap program are included in the documentation.

2.2 Software Installation

Place the CD-ROM in your drive. Go to the **Rigel Products | 166 Software | READS166 | READS310 .exe** program**.** Click on the exe file and the program will begin to load in your system.
Follow the standard install directions answering the questions with the appropriate answers.

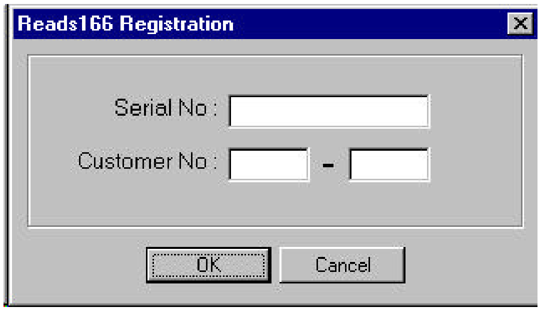
2.3 Start up

1. Connect your RIGEL board to the PC host via a serial cable.
2. Connect the board to a well-regulated 5-volt power supply.
3. Check to make sure the correct jumpers are in place.
4. Run the READS166 host driver by selecting **Start | Programs | READS166V3.10**. You may also start READS166 by double clicking on the READS166 short cut icon if installed.

2.4 Serial Number

2.4.1 Demo Version

When you run the READS software a pop-up registration box will open asking for your serial and customer numbers. If you are using the demo version of the software you may press **CANCEL, the box will disappear,** and the About Box will appear. Press the **OK** button in the About Box and the software will run in the Demo mode. The demo mode limits the size of the files you can compile to about 8K.



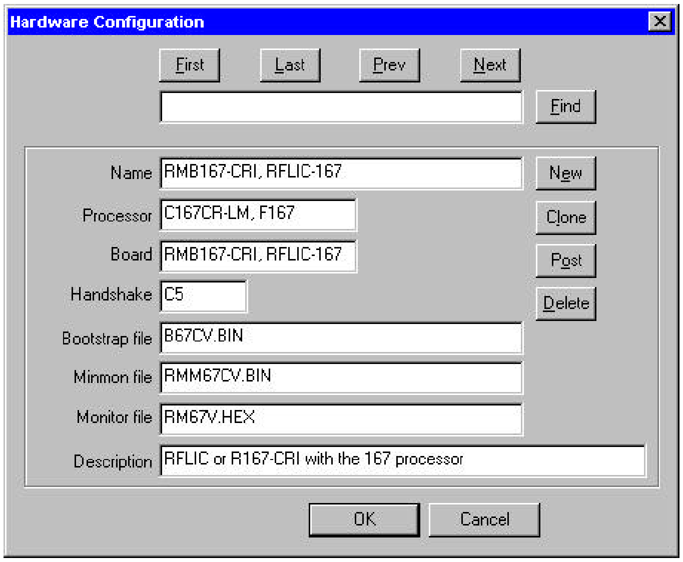
2.4.2 Full Version

When you run the READS software for the first time a pop-up registration box will open asking for your serial and customer numbers. Both numbers will be on a registration sheet and included with the printed documentation. Insert these numbers in the correct boxes, press **OK** and then press **OK** in the About Box. Your software is registered and ready to use.

2.5 Configuring READS166

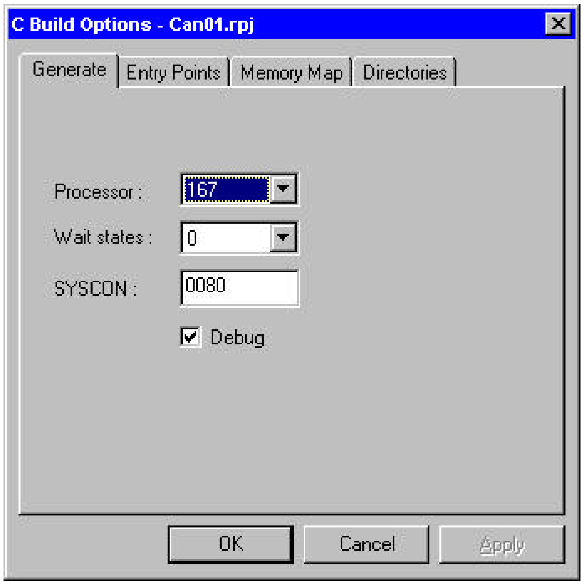
Please note that there are multiple ways to perform every basic operation available in the READS166 Software. Below are three different ways to select the board and processor you will be writing code for.

1. Using the **Tools | Hardware Configuration** menu command choose the name of the board you are using from the board list that appears.
2. Or you may open a project, and then from the **Projects | Project Build Options | Compile Options** | or **| Assembly Options |** select the processor you are using from the list.
3. Or from the **Options |Hardware Options**



Name RMB167-CRI, RFLIC-167
Processor C167CR-LM, F167
Board RMB167-CRI, RFLIC-167
Handshake C5
Bootstrap file B67CV.BIN
Minmon file RMM67CV.BIN
Monitor file RM67V.HEX
Description RFLIC or R167-CRI with the 167 processor

| you may select the board you will be using.

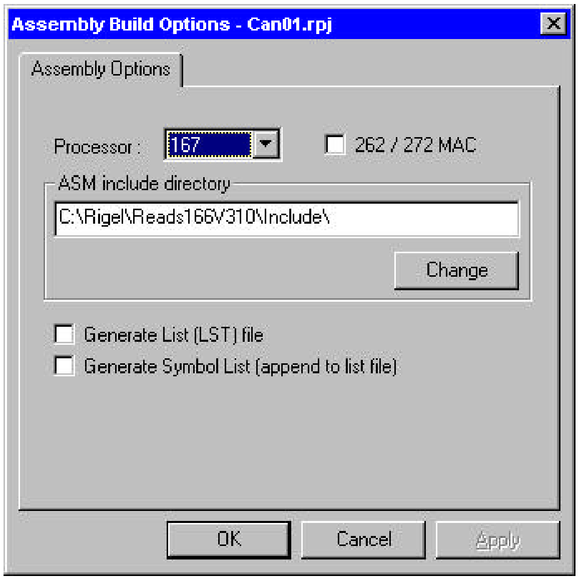

C Build Options - Can01.rpj

Processor : 167

Wait states : 0

SYSCON : 0080

☑ Debug


Assembly Build Options - Can01.rpj

Processor : 167

□ 262 / 272 MAC

ASM include directory: C:\Rigel\Reads166V310\Include\

□ Generate List (LST) file

□ Generate Symbol List (append to list file)

With an open project, the **C Build Options** and **Assembly Options** also allow you to automatically select from a variety of choices used when writing programs.

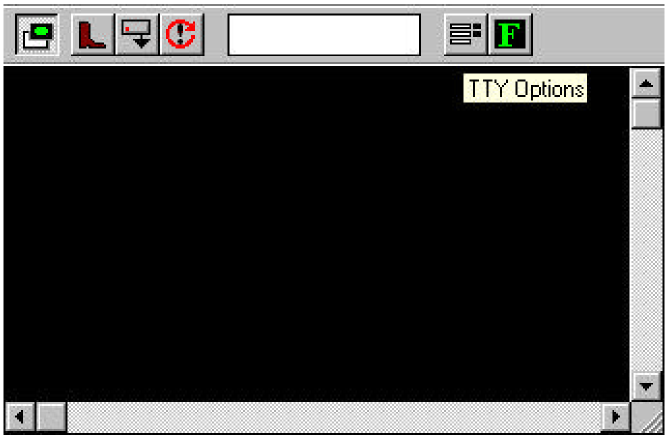
2.6 Initiating Host-to-Board Communications

Initiating communications between the PC and board may be done several different ways. Open the TTY window using the **Tools | TTY** menu command,

(or click on the smiley TTY face.)

Select the communication port parameters using the **Tools | TTY | TTY Options** menu command.

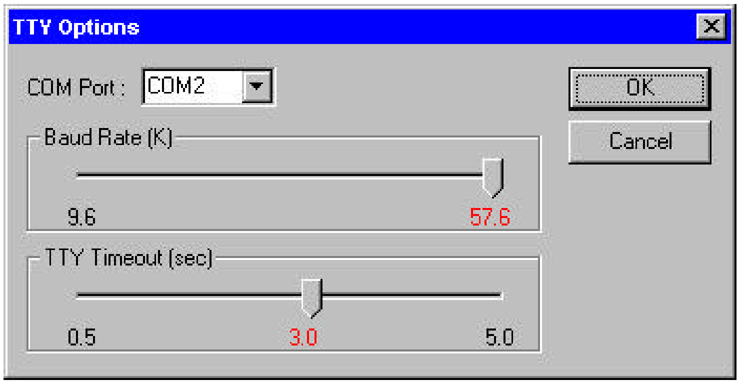
(Or select the options ☰ button in the TTY window.) You will need to select the COM port you are using, and the baud rate. You may also select the TTY Options from the **Options | TTY Options** | menu command.



2.7 Bootstrapping

In the default configuration, all monitor programs are downloaded to the boards after the boards are bootstrapped. That is, there is no ROM on the board that is executed upon reset. Bootstrapping loads a small monitor, called MinMon, which in turn loads a larger monitor RMON16x. Once the monitor program is loaded, the monitor commands are available to the user.

1. Press the reset button on the board.
2. From the menu in the TTY window select **TTY | Bootstrap and Download Monitor**. The board will now bootstrap and download the monitor program.


TTY Options

COM Port : COM2

Baud Rate (K): 9.6 – 57.6 (set to 57.6)

TTY Timeout (sec): 0.5 – 5.0 (set to 3.0)

3. Or select the boot in the TTY window.

You may observe the bootstrap progress in the status line of the TTY window. When bootstrapping is completed, the READS166 monitor prompt appears in the TTY window.

2.7 Verifying that the Monitor is Loaded

Make sure the TTY window is active, clicking the mouse inside the TTY window to activate it if necessary. Then type the letter '***H***' (case insensitive) to verify that the monitor program is responding. The 'H' command displays the available single-letter commands the monitor will recognize.

The READS monitors use single-letter commands to execute basic functions. Port configurations and data, as well as memory inspection and modifications may be accomplished by the monitor. Most of the single-letter commands are followed by 4 hexadecimal digit addresses or 2 hexadecimal digit data bytes. The following is a list of the commands.

READS COMMANDS

C nn	read port nn Configuration (DPnn)
C nn=mmmm	set port nn Configuration (DPnn=mmmm)
D	Download HEX file
G XXXX	Go, execute code at XXXX
H	Help, display this list
M XXXX	Memory, contents of XXXX
M XXXX=nn	Memory, change contents of XXXX to nn
M XXXX-YYYY=nn	Memory, change block XXXX-YYYY to nn
P nn	read Port nn (Pnn)
P nn=mmmm	write to Port nn (Pnn=mmmm)
W XXXX	Word memory, contents of XXXX
W XXXX=mmmm	Word memory, change contents of XXXX to mmmm
W XXXX-YYYY=mmmm	Word memory, change block XXXX-YYYY to mmmm

3.0 READS166 V3.10 CONCEPTS

READS166 introduces a project-oriented code development and management system. The concepts are defined below.

3.1 Project

A project is a collection of files managed together. Each file in a project corresponds to a code module. All projects are kept in their individual subdirectories. You may copy or save projects as a single entity. When saved under a different name, a new subdirectory is created and all components of the project are duplicated in the new subdirectory.

By using the long names provided by the 32-bit Windows operating systems, you may use this feature to keep different versions of your software in a controlled manner. For example, the project “Motor Control 07-20-1997” may be saved under the name “Motor Control 07-25-1997” as new features are added. This way, if needed, you may revert to an older version.

A project may either be an “executable project” or an “archive project.”

3.1.1 Executable Projects

Executable projects are meant to be compiled into code that is eventually run on the target system. Components of an executable project are the code modules containing subroutines or functions, which make up the entire program.

3.1.2 Archive Projects

Archive projects are never compiled. They are intended to facilitate code reusability by organizing and keeping code modules together. An archive project acts as a repository that you may add modules to, or copy modules from. Executable projects can be quickly constructed using already written and debugged modules from an archive project.

3.2 Module

A module is a single file that belongs to a project. Typically modules are either assembly language subroutines or C language functions. You may copy modules from one project to another, or share modules in different projects. For example, you may copy a previously developed module from an archive project to an executable project by simply dragging its icon from one project window to the other. By using existing or previously developed and debugged modules, you may significantly improve code reusability, much in the same manner as libraries. Reusing modules differs from using library functions of existing routines in that modules are kept in source form rather than object form.

4.0 TUTORIAL 1 -- EXECUTABLE PROJECTS

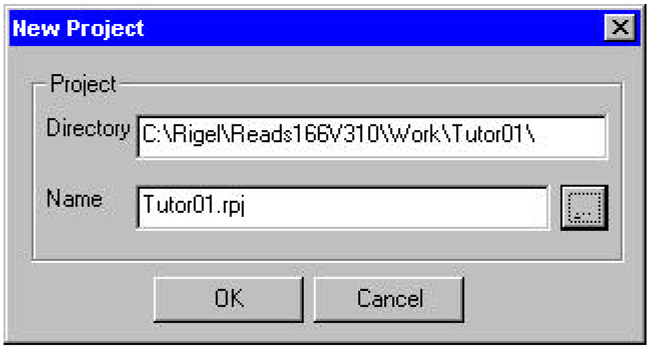
A project is a collection of code modules that are grouped together. READS166 uses two different kinds of projects, executable projects and archive projects. An executable project is a collection of modules which are compiled together to form one executable program. An archive project is a collection of modules that are grouped together to make retrieval from storage easier.

We will first create an executable project. We will explain creating archive projects in a later section.

4.1 Creating an Executable Project

1. Select **Project | New Project | Executable Project**

2. A New Executable Project box will open up.



New Project

Project

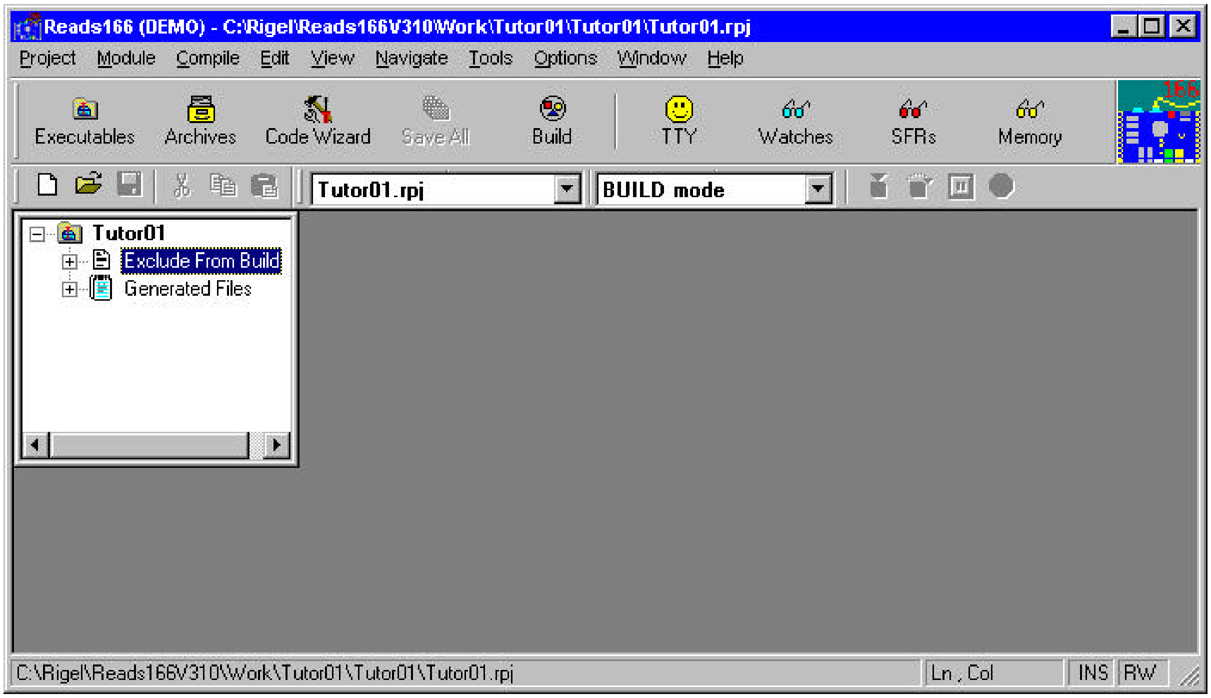
Directory C:\Rigel\Reads166V310\Work\Tutor01\

Name Tutor01.rpj

3. In the **Name** box type *Tutor01*. You may also browse the available selections by clicking on the … icon and looking in the work directory. Tutor01 is already in the work directory.

4. Click **OK**

5. A project window box will open which shows *TUTOR01.RPJ* represented as the root of a tree. Next we will add a module to it.



Reads166 (DEMO) - C:\Rigel\Reads166V310\Work\Tutor01\Tutor01\Tutor01.rpj

Tutor01.rpj

BUILD mode

Tutor01
Exclude From Build
Generated Files

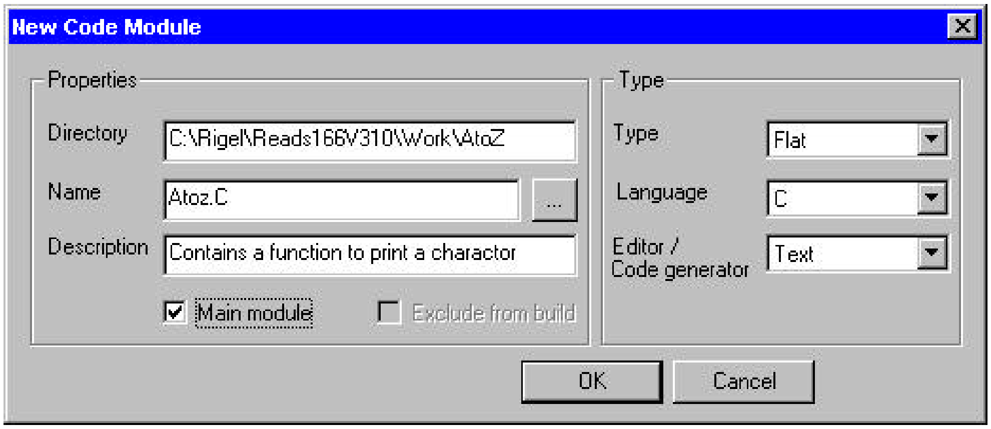
C:\Rigel\Reads166V310\Work\Tutor01\Tutor01\Tutor01.rpj

4.2 Adding a Module

A module is a block of code that is saved in a single source file. To add a module to the executable project *TUTOR01.RPJ*, make sure that the project window is open and double click on *TUTOR01.RPJ*. If the project window is not open, select **Project | Open Project | Executable Project** and double click on *TUTOR01.RPJ*. Now, you can add a module.

1. Select **Module | Add Module**. A New Code Module window will appear.

2. Type *AtoZ.c* in the **Filename** box, click on the … icon and select the AtoZ.C module from the list.



Directory: C:\Rigel\Reads166V310\Work\AtoZ

Name: Atoz.C

Description: Contains a function to print a charactor

☑ Main module □ Exclude from build

Type: Flat

Language: C

Editor / Code generator: Text

3. In the **Description** box type *Contains a function to print a character*

4. Click **OK**

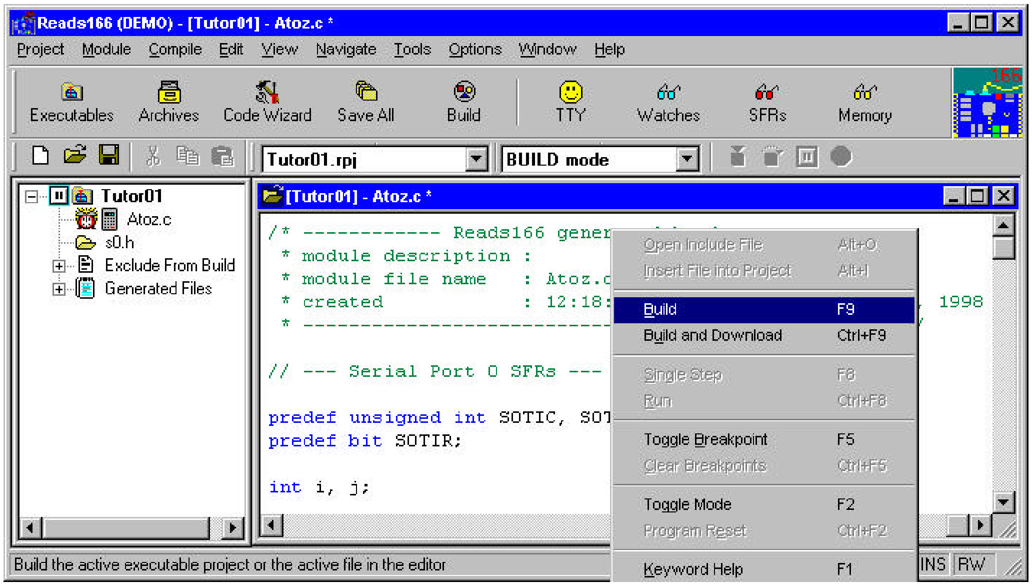
ATOZ.C has been added. It appears as a branch of the project *TUTOR1.RPJ*. If you move the mouse over *ATOZ.C* you will see its description in the status bar. This operation makes *TUTOR1.RPJ* a *C* project.

The language box in the New Code Module window shows that the module will be in C. The language was automatically selected when you typed in AtoZ with the extension .c. If you type in the same module name AtoZ with the extension .asm the language box will automatically select the language to be assembly.

4.3 Building the Project

1. Double click on the ATOZ.C module. A READS Editor box with the program opens up.

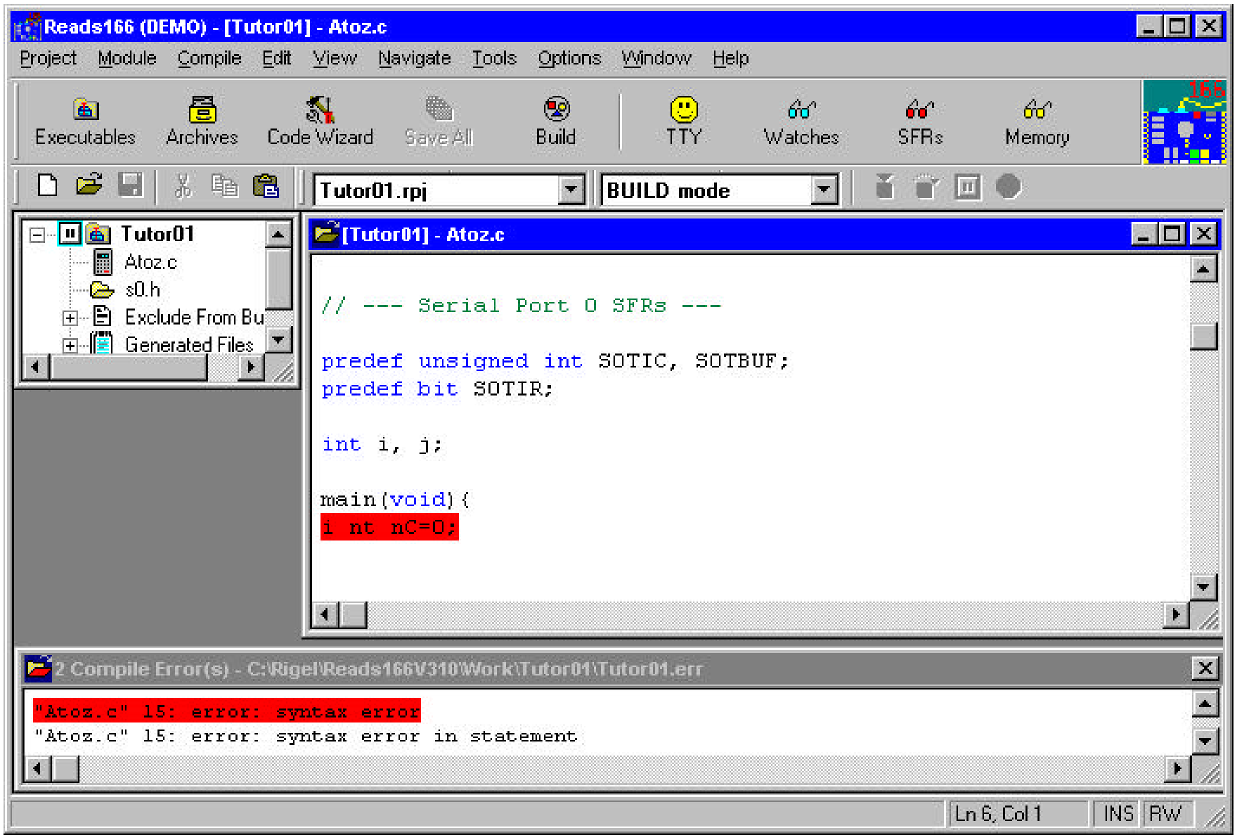
2. Click on the right mouse button and select **Build** from the sub-menu items displayed.



```
/* ------------ Reads166 gener
 * module description :
 * module file name   : Atoz.c
 * created            : 12:18:                1998
 * ---------------------------

// --- Serial Port 0 SFRs ---

predef unsigned int SOTIC, SOT
predef bit SOTIR;

int i, j;
```

Build the active executable project or the active file in the editor

3. A READS Error Dialog box will open up which will report a syntax error in line 15.


Reads166 (DEMO) - [Tutor01] - Atoz.c
Project Module Compile Edit View Navigate Tools Options Window Help
Executables
Archives
Code Wizard
Save All
Build
TTY
Watches
SFRs
Memory
Tutor01.rpj
BUILD mode
Tutor01
Atoz.c
s0.h
Exclude From Bu
Generated Files
[Tutor01] - Atoz.c
2 Compile Error(s) - C:\Rigel\Reads166V310\Work\Tutor01\Tutor01.err
Ln 6, Col 1
INS
RW

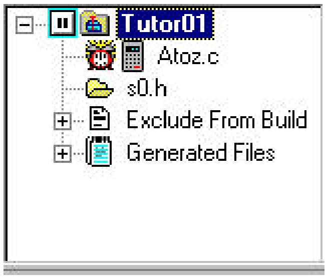
```
// --- Serial Port 0 SFRs ---

predef unsigned int S0TIC, S0TBUF;
predef bit S0TIR;

int i, j;

main(void){
i nt nC=0;
```

```
"Atoz.c" 15: error: syntax error
"Atoz.c" 15: error: syntax error in statement
```

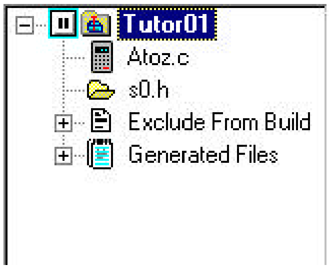
4. Line 15 will be highlighted in the READS Editor box.

5. Correct the highlighted syntax error by removing the space between the *i* and the *nt.*

6. Notice in the project window that an alarm clock appears next to the open module ATOZ.C. This symbol shows the module has been modified but not saved. If you click on the alarm clock the file will be saved.



File Modified But Not Saved



File Saved

7. Right click and select **Build** in the READS editor box again.

8. A Build box will appear which shows the ATOZ.C file was modified. The **Save modified files and build** option is already selected.

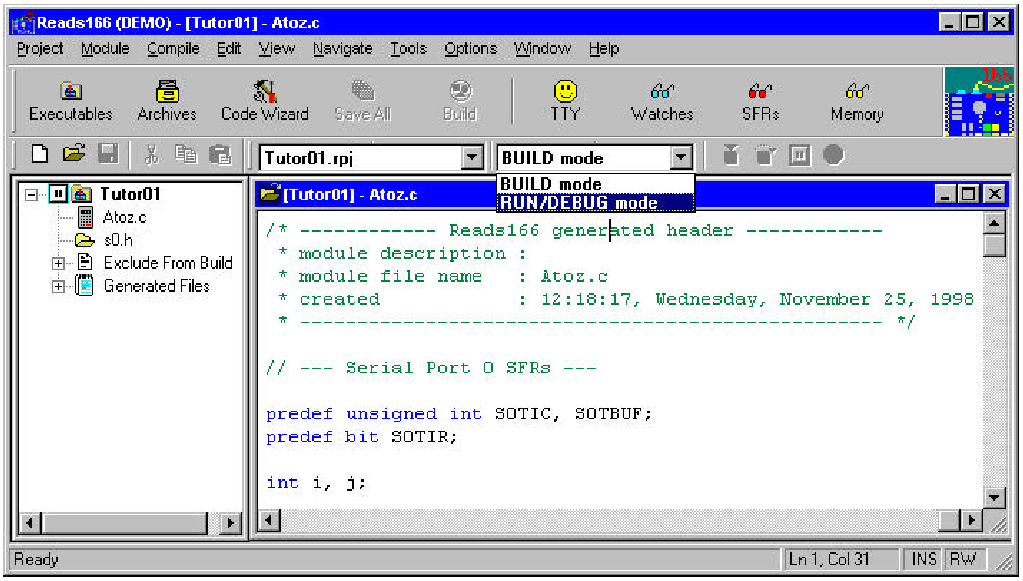
9. Press **OK** at the **Build** dialog. Notice the alarm clock next to the to the open module ATOZ.C has disappeared, indicating the module was saved.

TUTOR01.HEX has just been created and it is ready to download and run.

4.4 Running the Project

Running the project *TUTOR01.RPJ* requires you to switch to the **Run / Debug mode**. In Run / Debug mode *TUTOR01.HEX* is automatically downloaded to the board. Make sure that one of the serial ports on your computer is connected the board and correct port parameters are entered in **Options | TTY options.**

1. Click the **Run / Debug mode** on the toolbar.



```
/* ------------ Reads166 generated header ------------
 * module description :
 * module file name   : Atoz.c
 * created            : 12:18:17, Wednesday, November 25, 1998
 * --------------------------------------------------- */

// --- Serial Port 0 SFRs ---

predef unsigned int S0TIC, S0TBUF;
predef bit S0TIR;

int i, j;
```

2. If the board is not already bootstrapped, a message box will appear asking for permission to bootstrap the board. Press the ***RESET*** button on the board and click Yes.

3. The TTY window will open and the board will be bootstrapped and the monitor program and Tutor01 project will be downloaded to the board.

4. When the program has finished downloading to the board the monitor prompt will appear in the TTY window,

5. Select the **Compile | Run** option. The program will then run on the board.

6. A string of characters from *A* to *Z* will appear in the TTY window.

7. The TTY window will indicate when the program has finished running by returning the monitor prompt.

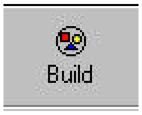
5.0 TUTORIAL 2 -- DEBUGGING A PROJECT

To debug the executable project *TUTOR1.RPJ*, the debug option must be selected before the project is built and downloaded to the board. To select the debug option follow these steps.

5.1 Selecting the Debug Mode

1. To select the debug mode, first make sure that the project is open. If not, select **Project | Open Project | Executable Project** and double click ***TUTOR01.RPJ***.

2. Select the **Build** mode from the toolbar.



3. Right click on the ***TUTOR01.RPJ**,* and click on the **Options | Compile options**. (You can also make this selection using the main menu and selecting the **Project | Project Build Options | Compile Options**.)

4. A C Compiler Options box will appear.

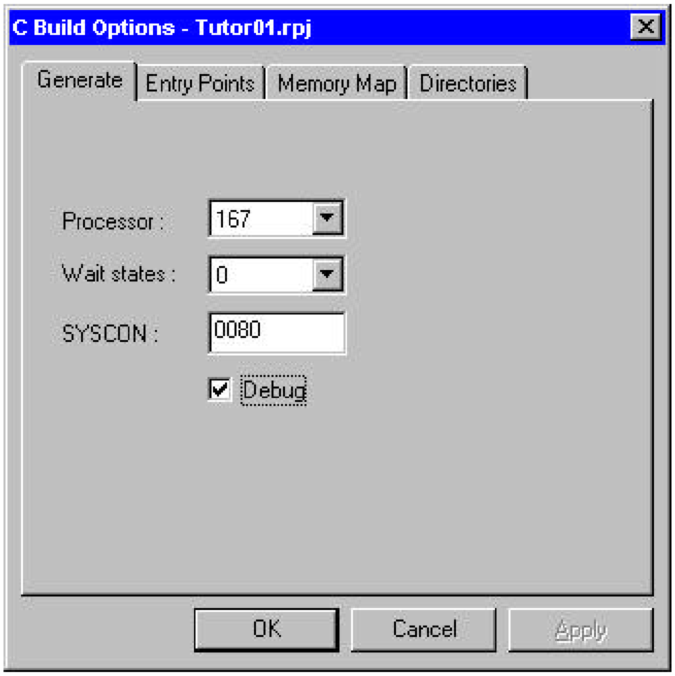
5. Select the **Generate** tab

6. Check the **Debug option**

7. Click **OK**

8. Select **Tutor01.rpj** by clicking on it. Then right click and select **Build** from the list. The project will be compiled and ready to run.

9. Select the **Run | Debug** mode from the toolbar.



Processor : 167

Wait states : 0

SYSCON : 0080

☑ Debug

TUTOR01.HEX is downloaded to the board and ready to debug.

5.2 Adding Breakpoints and Singlestepping

1. Double click the module **ATOZ.C** to open the READS Editor box.

2. In the editor go to the line where *SendChar(nC)* is written and set the cursor to point to this line.

3. Press **F5**, to set a breakpoint at this line. The line *SendChar(nC)* turns blue indicating that a breakpoint has been set.

4. Press **Ctrl F8**. The program will run until it reaches the breakpoint

5. Press **F8** to start singlestepping through the program.

6. Press **F8** again, the character ***A*** has been printed in the **TTY** window.

7. Press **Ctrl F8** again. The program will run until it reaches the breakpoint.

8. Continue to press **Ctrl F8.** Each time you will stop at the breakpoint and a new letter will be added in the **TTY** window.

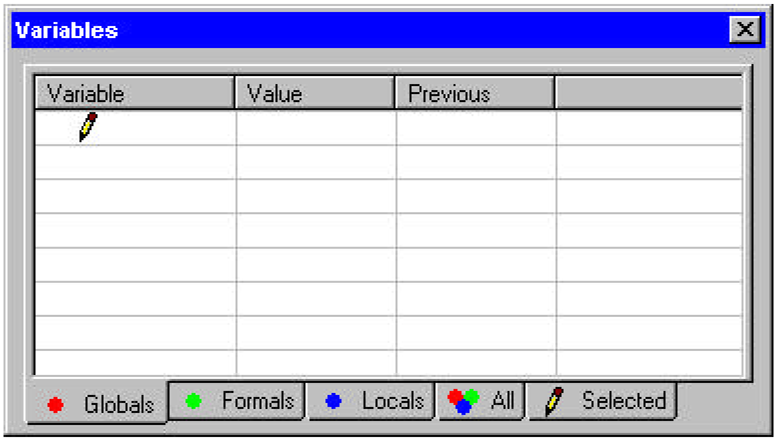
9. Singlestep to the line where you have set the breakpoint and press **F5** to clear the breakpoint.

10. Press **Ctrl F8** to run the program until it finishes or press **Ctrl F2** to reset

5.3 Watching Variables

READS166 allows the user to watch the variables as the program is debugged.

1. Select **View | Watches** or the button from the tool bar.



2. Select the **Globals** tab in the Watches window. You will see all the variables including the formal parameters

3. Press **F8** to start singlestepping

6.0 TUTORIAL 3 -- ARCHIVE PROJECTS

Archive projects are intended to facilitate code reusability by organizing and keeping code modules together. An archive project acts as a repository that you may add modules to, or copy modules from. Executable projects can be quickly constructed using already written and debugged modules from an archive project.

6.1 Creating an Archive Project

An archive project is used to store source code that may then be used for other projects. READS166 uses the Drag and Drop method to of moving modules from place to place. To create an archive project:

1. Select **Project | New Project | Archive Project**

2. Type ***Archive*** in the **Name** box

3. Click **OK**

Now you will see the project window *ARCHIVE.RAR* represented as the root of a tree.

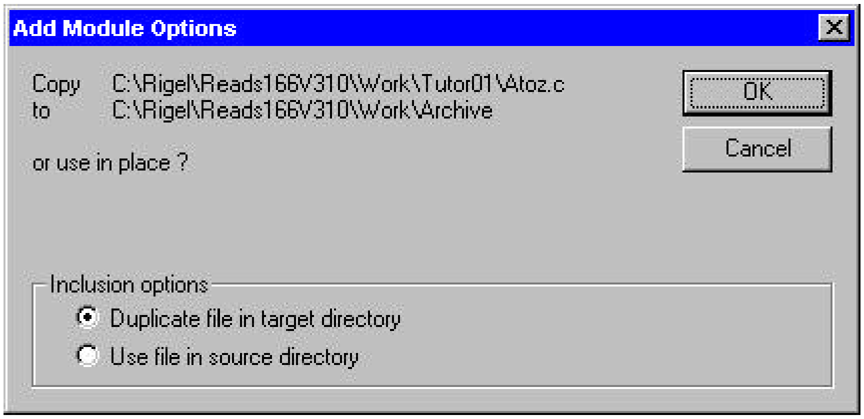
6.2 Importing/Exporting Modules

1. Open the executable project TUTOR1.RPJ by selecting **Project | Open Project | Executable Project** and double clicking **TUTOR1.RPJ**

2. Open the archive project ARCHIVE.RAR by selecting **Project | Open Project | Archive Project** and double clicking **ARCHIVE.RAR**

3. Click on the module **ATOZ.C** and hold the mouse button down while you are dragging toward ARCHIVE.RAR. When the archive project name is highlighted release the mouse button.

4. Click **OK** meaning that you want to make duplicate of ATOZ.C under the archive directory.



Copy C:\Rigel\Reads166V310\Work\Tutor01\Atoz.c
to C:\Rigel\Reads166V310\Work\Archive

or use in place ?

Inclusion options

● Duplicate file in target directory

○ Use file in source directory

You've created your own archive file containing the module AtoZ.C. You may add as many modules to an archive project as you want. Usually the modules you will want to add are tested and debugged modules that you may want to use for other projects. READS166 includes many of the more common modules in C and Assembly for the A/D routines, timers, counters and so on.

For more on building projects and writing your own programs see the tutorial in the READS166 on-line HELP.

7.0 CODE WIZARDS

7.1 Introduction

Code generators are a recent effort in the quest for code reusability. Code generators present a good approach to create specific functions for user-specified operating modes. The READS166 software currently has three code generators or wizards available. The three code wizards are for ADC, software timers, and state machines. Rigel will continue to add code wizards to the software and new wizards may be downloaded from the web site, as they become available.

With the new READS166 environment the end user has the option of writing their own code wizards. When a C code wizard is dragged-and-dropped into a project, the integrated development environment (IDE) calls the Reads166 code wizard. This wizard has a “Setup” button that allows users to insert other code generators into the existing list. For a custom code generator to be recognized, it must be written as a Windows Dynamic Link Library (DLL). There is article on the Rigel web site, which explains and gives an example of how to write your own code wizards.

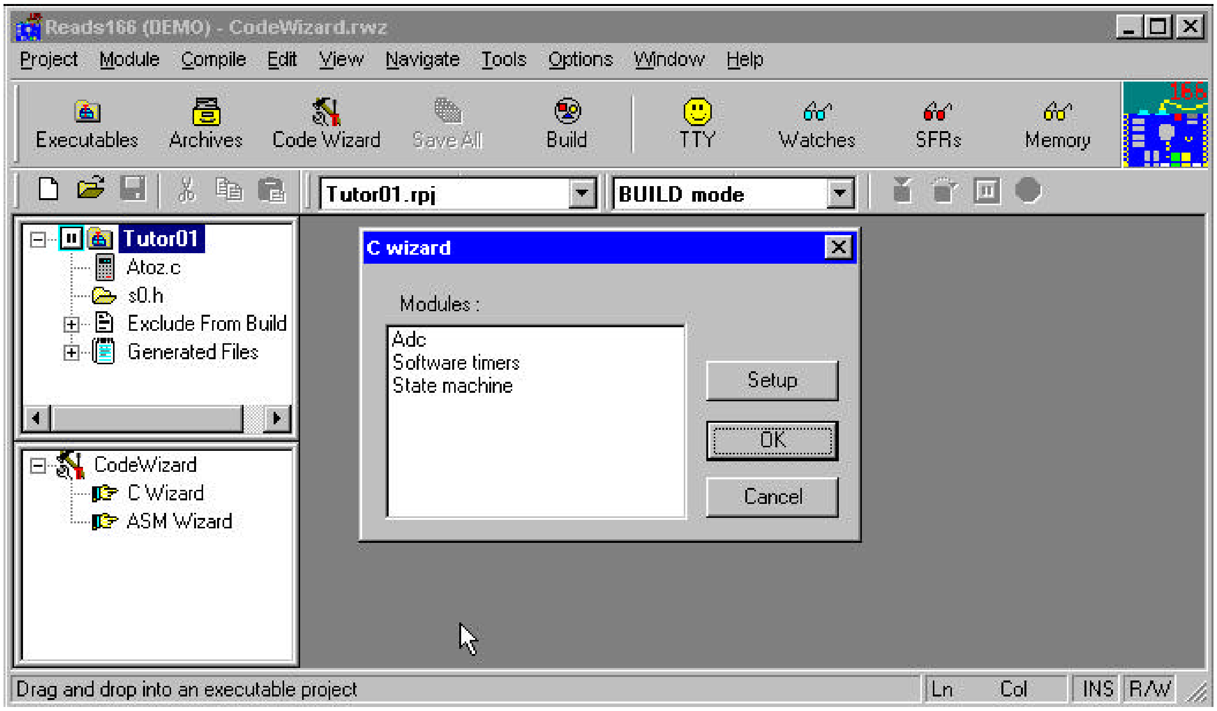
7.2 Using the Code Wizards

Code wizards may be used in any open project to speed code development. Currently three code wizards are available in the READS166 software. The wizards will generate code for ADC, software timers, and state machines.

1. To activate a code wizard an executable project must be open.

2. Select the code Code Wizard wizard button on the tool bar. A code wizard window will open below the current project.

3. Click on the C Wizard and drag it into your current project. A C wizard box will open which allows you to select what kind of module you’d like to add to your project.

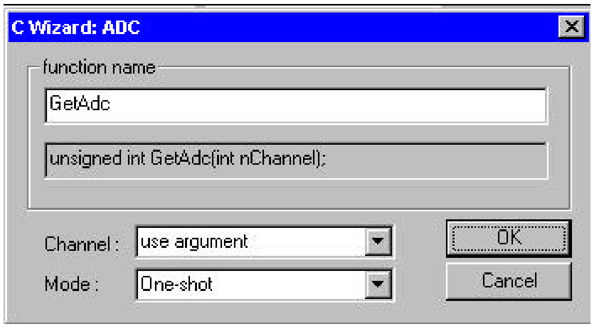


Modules :

Adc
Software timers
State machine

Drag and drop into an executable project

5. Select the module you want and another window will open which allows you to input your choices for generating the code.


C Wizard: ADC

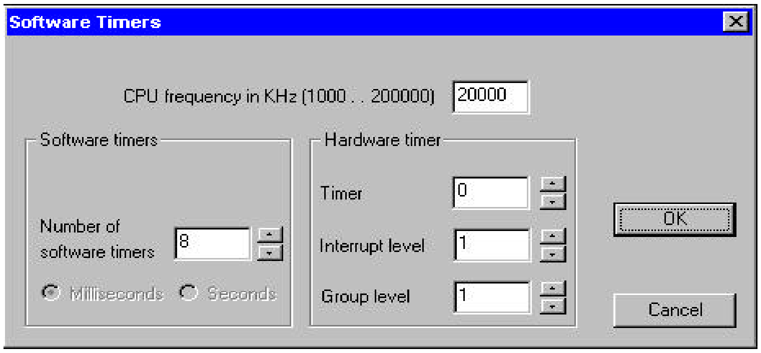
function name

GetAdc

unsigned int GetAdc(int nChannel);

Channel : use argument

Mode : One-shot


Software Timers

CPU frequency in KHz (1000 . . 200000) 20000

Software timers

Number of software timers 8

● Milliseconds ○ Seconds

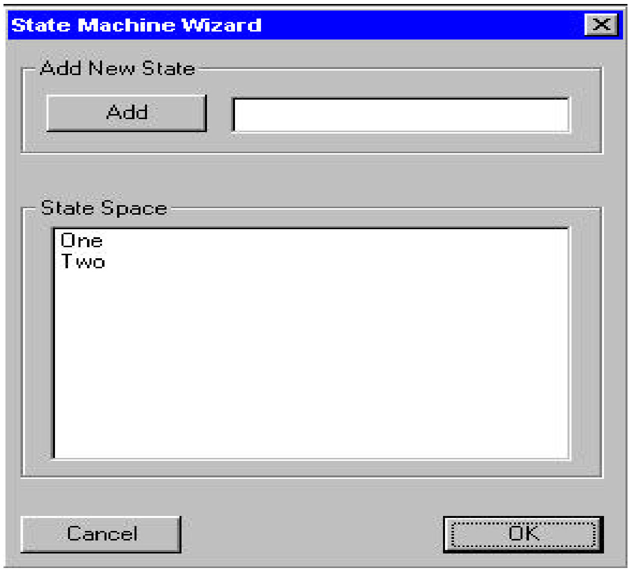
Hardware timer

Timer 0

Interrupt level 1

Group level 1

6. Once you've inputted your choices press okay and a new module will be added to your project.


State Machine Wizard

Add New State

State Space

One
Two

8.0 Ra66 -- THE READS166 ASSEMBLER

8.1 Overview of Ra66

Ra66 is a cross assembler for the C166 family of microcontrollers. It is intended to be used by the hardware and software products available from Rigel Corporation. Ra66 is a two-pass assembler. Forward references are resolved during the second pass. The second pass is used only when necessary. If no forward references are used, Ra66 completes assembly in a single pass.

8.2 Pseudo Operations

There are nine pseudo-operations (pseudo-ops) recognized by Ra66. These are DB (define byte), DW (define word), DS (define segment of bytes), DSW (define a segment of words), EQU, DEFR, DEFB, DEFA, and ORG (origin) pseudo-operations.

8.3 Types and Typecasting

Types are properties of the operands. Strictly speaking, the assembly language of C166 family of microcontrollers restricts the use of certain types of operands in certain instructions. As a design philosophy, Ra66 does not enforce strict type checking. Strict type checking has advantages in writing high-level language compilers which generate assembly code. Type checking, however, may become somewhat awkward when writing small assembly routines by hand. For example, Ra66 with only a few exceptions allows the use of constants defined by EQU pseudo operations in all instructions.

Perhaps the most important type designation is the size of data or memory: BYTE or WORD. Byte and word operations are performed by different instructions. For example, AND is a word operation while ANDB is its byte-wise counterpart. Since the type of operand (byte or word) is implied by the instruction, the type of the operands may be overlooked, and the operation may be performed assuming the operands are of the type implied by the instructions.

There are two other operand types besides BYTE and WORD, namely BIT and BITWORD. The latter is a word of the bit addressable memory. Similarly, constants have types that indicate their size in terms of the number of bits. DATA3, DATA4, DATA8, and DATA16 are the allowed types of constants. Finally, pointers to code are classified as SHORT, NEAR, and FAR. SHORT is used when the code is within -127 to 128 words of the current address. All conditional relative jumps assume that the operand is a memory address of type SHORT. NEAR pointers may be located anywhere within the current 64K segment. FAR pointers are used with inter-segment jumps and calls. In addition, the keyword PTR may be used in conjunction with types BYTE and WORD.

General operand types:	BIT	BYTE	BITWORD	WORD
Constant types:	DATA3	DATA4	DATA8	DATA16
Pointer types:	NEAR	FAR	SHORT	

Typecasting refers to the explicit naming of the operand types. Consider, for example, the constant value being used as a pointer to code.

```
value EQU   1234h
calla cc_UC, NEAR 1234h
```

The absolute call instruction expects an operand of type NEAR. However, value is defined by an EQU pseudo operation. By default, value is a constant of type DATA16. The instruction above explicitly typecasts value from type DATA16 to NEAR. As stated, Ra66 is often very forgiving and does not generate an error for such type mismatches. In certain cases, especially when it leads to ambiguities, explicit typecasting may be required.

Next, consider the instruction,

```
movb  RL0, 1234h
```

which moves a single byte at memory location 1234h to register RL0. The second operand is the address of a byte. However, as such, the assembler reads 1234h as a simple constant of type DATA16. A strictly type-checking assembler would require the typecasting 1234h to a byte address as below.

```
movb  RL0, BYTE PTR 1234h
```

Again, Ra66 allows typecasting but does not enforce it as long as the type of an operand is implied by the instruction.

The ORG pseudo-op sets the location counter to the specified value. The location counter is the current address. The EVEN compiler directive inserts a single byte if the current location counter is an odd number. This assures that the next instruction starts at an even address.

DB, DW, DS, and DSW allocate up to 255 bytes of memory. The defined block of data may optionally be initialized with the DS and DSW. The allocated memory block is placed at the current location counter. The ORG compiler directive should be used if the memory block is to be placed at a specific address. Labels preceding the DB, DW, DS, or DSW pseudo-ops provide a convenient means to address the data bytes or words.

The syntax for each pseudo-op is illustrated by the examples below.

```
DATA1:
      db   0,'A',2+3          ; sets aside 3 bytes at the current address.
                              ; This address may be referred to by the label
                              ; DATA1.  The bytes are initialized to 0, 41h,
                              ; and 5

DATA2:
      dw   0,'A', 1234h       ; sets aside 6 bytes at the current address. The
                              ; bytes are initialized to 0, 41h and 1234h

DATA3:
      ds   10                 ; sets aside 10 bytes at the current address.
                              ; All 10 bytes are initialized to 0

DATA4:
      ds   10 (101b)          ; sets aside 10 bytes at the current address.
                              ; All 10 bytes are initialized to 101b or 5

DATA5:
      dsw   10h               ; sets aside 10h=16 words (32 bytes)at the
                              ; current address. All 16 words are initialized
                              ; to 0

DATA6:
      dsw   10h (3)           ; sets aside 10h=16 words or 32 bytes at the
                              ; current address.  All 16 words are initialized
                              ; to 3
```

Expressions may be used in specifying the contents of the memory block. For example,

```
BEGIN:
            .
            .
            .
DATA7:
      dw   begin
```

places the value of the label BEGIN at the current location. DATA7 may be interpreted as a pointer that initially points to BEGIN. This pointer may be changed later by the program to point to another label.

The pseudo-op EQU, DEFA, DEFB, and DEFR, define symbols. EQU is used for generic symbols, DEFA, for internal RAM locations, DEFB for bits of the bit addressable internal memory, and DEFR, for the Special Function Registers (SFR) of the microcontroller. SFRs must be defined at even addresses. A dot (.) is used to separate the byte address from the bit in defining bits. Some examples follow.

```
SFR1        DEFR  0ffe0h
SFR2        DEFR  0ffe2h
RAM1        DEFA  0faffh
Flags DEFA  0fdd0h
Flag1 DEFB  0fdd0h.0          ; bit 0 of byte 0fdd0h
Flag2 DEFB  Flags.1           ; bit 1 of byte 0fdd0h

Speed EQU   1234h
HiSpeed     EQU   2*Speed     ; HiSpeed is 2468h
```

8.4 Compiler Directives

There are two compiler directives: #INCLUDE and EVEN. The include directive inserts another file into the source. It is used as below.

```
#include "filename.ext"
```

The file specified by filename.ext must be found in the current directory or path. The file filename.ext will be opened and merged with the source (assembly) code. The number of include files is limited by the DOS parameters BUFFERS and FILES. See your DOS manual for further details. Up to 8 include file may be nested. That is, include files may be specified inside include files, nested up to 8 levels.

The EVEN compiler directive inserts a single byte if the current location counter is an odd number. This assures that the next instruction starts at an even address.

8.5 Constants

Constants may be decimal, binary, hexadecimal, or ASCII. Hexadecimal constants must start with a numerical digit between 0 and 9. The suffixes b, d, and h are used to denote binary, decimal, and hexadecimal constants, respectively. If no suffix is present, the constant is assumed to be decimal. ASCII constants are placed within single quotation marks. See the following examples.

10, 10d, 10D, 0ah, 0aH, 0Ah, 0AH, 1010b, 1010B all have the value 10.

Note that hexadecimal numbers must start with a numerical character. Although 0ah is valid, ah is not.

The ASCII code of the letter 'A ' is 41h. Ra66 converts 'A' to the numerical value 41h. This numerical value may then be used in expressions, such as 'A'+3, which is evaluated as 44h.

8.6 Expressions

Expressions may involve constants, defined values, or label values (memory addresses). During assembly passes, expressions are evaluated as soon as all the information is available. Thus, expressions that contain forward references are evaluated during the second pass. Expressions may require looking up the equate tables recursively. For example, in evaluating the value of `four` the assembler must evaluate the symbol `two`, and then the symbol `one.`

```
four  EQU two*two
two   EQU one+one
one   EQU 1

mov   RH0, #four
```

The following basic logical and arithmetic operations are supported.

Operation	Symbol	Alternate Symbol	Remark
Add	+		
Subtract	-		
Multiply	*		
Divide	/		
Modulus	%	MOD	
Invert	!	NOT	
Shift Left	<<	SL	parentheses required
Shift Right	>>	SR	parentheses required

Parentheses may be used to group terms of an expression. The parentheses may be nested. The number of such nestings is limited only by the amount of dynamic memory available. Some examples follow.

```
1+2         ; 1+2 is 3
3*(1+4)     ; 3*(1+4) is 15
1010b/10b   ; 10/2=5
(1<<2)      ; 1<<2 is 4.  Parentheses required.
(1 SL 2)    ; same as 1<<2
(0F0h>>4)   ; 0F0h>>4 is 0Fh=15. Parentheses required.
(1 SR 2)    ; same as 0F0>>4
5%2         ; 5 % 2 is 1
5 MOD 3     ; the same as 5%2
!5          ; !5 is FFFAh (or FAh if the type is BYTE or DATA8)
NOT 5       ; same as !5
```

8.7 Functions

There are two built-in functions: high() and low(). They return the high byte and the low byte of a word (two-byte expression), respectively.

```
Low(1234h)   is 34h
High(1234h)  is 12h
```

8.8 Addressing Modes

The instructions use a powerful set of addressing modes. The operands are usually one of REG, MEM, DATA, or Rw (GPR) types. REG is a Special Function Register (SFR) which, in addition to its 16-bit absolute address, has an 8-bit number. MEM is the address of a memory location, DATA is a constant of 3, 4, 8, or 16 bits, and Rw is a register of the register bank. Word registers are denoted by Rw, and the byte registers by Rb. The constant data operands typically follow a pound sign '#'. This implies that the operand is a constant value rather than a memory location. For example, both of the instructions

```
mov   R0,   #1234h
mov   R0,   1234h
```

place a word (two-byte value) in the R0 register. In the first case, the value 1234h is placed into the register, 12h as the high byte, and 34h, as the low byte. The second instruction moves the contents of the memory location 1234h into the register R0. Thus, the actual content of R0 after the instruction depends on exactly what is in memory location 1234h at run time.

The examples given below refer to variables (word and byte memory locations), B1 and W1, as well as bits BIT1 and BIT2. These variables may be defined by the DB, DW, and DEFB pseudo-operations as follows.

```
; -----------------------------------------------------
B1:    db 0
       EVEN
W1:    dw 0
BIT1   DEFB  0FDDEh.0
BIT2   DEFB  0FDDEh.1
; ------------------------------------------------------
```

See the demonstration program DEMO02.asm for a list of all instructions.

8.9 Assembly Errors

8.9.1 Attempt to Redefine Symbol or Label

A label or symbol of the same name was previously defined.

8.9.2 Incorrect Symbol or Label

Symbols and labels may only include letters [a-z], or [A-Z], digits [0-9], or the underscore character (_).

8.9.3 Incorrect Operand

The operand type is not permitted in the instruction. For example,

```
movb R0, R1
```

is a byte-oriented move, where the operands are word operands.

8.9.4 Attempt to Branch Out of Bounds

The jump point of a branching instruction is beyond reach. Relative jumps and calls are limited to the range of [-127 to 128] words from the current address. The current address is the address of the first byte of the following instruction. Note also that target addresses must be even, since all C166 instructions start at even addresses.

8.9.5 Unresolved Operand(s)

Either a typographical error was made in naming the operand, or the operand is not defined. If the operand is an expression, one or more of the terms is undefined.

8.9.6 Undecodable Line

This error is a "catch-all" error. Misspelled operation codes will generate this error. As in, for example,

```
       move R0, R1
```

Note that the assembler continues to read tokens until a valid operation code is detected. Therefore, this error may be given after the instruction following the "MOVE" instruction. That is, the assembler may assume that MOVE is a label or a symbol, for example.

8.9.7 Operand(s) Out of Range

This message is generated when the specified operand has a value too large or too small. For example,

```
       movb  RL0, #1234h
```

attempts to move a 16-bit word into a byte register.

8.9.8 Incorrect Operand Types

Some instructions are limited to word, byte, or bit operands. Moreover, a word may be a memory location, a Special Function Register address or a data byte of type #data16. Sometimes this error is generated when a symbol is not properly defined.

8.9.9 Incorrect Register Use

An operand, which is a constant or a memory type, was expected, but a register was found.

8.9.10 Incorrect Constant

A constant or an expression contains an error. For example, hexadecimal numbers must start with a numerical digit and end with the letter 'h' or 'H'. Expressions involving incorrect constants also generate this message.

8.9.11 Odd or Out-of-Range Address

The specified address is either odd or beyond the reach of a branching instruction. See the error message "Operand(s) Out of Range."

8.9.12 Undefined Symbol

A symbol appears in the instruction, but no definition of the symbol is found. Sometimes this message is generated if an include file containing the symbol definitions was not found, or when a misspelled operation code is mistaken for a symbol.

8.9.13 Incorrect Processor

There are some instructions specific to the C167, such as ATOMIC, which are not supported by the C166. Change your hardware configuration.

9.0 THE C166 INSTRUCTION SET

9.1 Overview

The instruction set of the C166 family of microcontrollers is classified into the following categories:

Arithmetic	
Logic	
Boolean	Bit-oriented instructions
Compare	
Loop control	
Prioritization	
Shift and rotate	
Data transfer	Move instructions
Branching	Call and Jump instructions
System Stack	Push and Pop instructions
System Control	

9.2 Arithmetic Instructions

Arithmetic instructions consist of the following:

- Addition
- Subtraction
- Multiplication
- Division
- Complement
- Negation

9.2.1 Addition

There are four addition instructions: ADD, ADDB, ADDC, and ADDCB. ADD and ADDC are used for word (16-bit) operands. ADDC adds the two operands and the carry flay. ADDC is convenient for multi-word addition tasks. ADDB and ADDCB are the addition instructions used for byte (8-bit) operands. The addressing modes are listed below.

ADD

```
add   R0, R1            ; Rw. Rw
add   R0, [R1]          ; Rw, [Rw]
add   R0, [R1+]         ; Rw, [Rw+]
add   R0, #0            ; Rw, #data3
add   SFR1, #0ffffh     ; reg, #data16
add   R0, #0ffffh       ; reg, #data16
add   SFR1, 1234h       ; reg, mem
add   SFR1, #W1         ; reg, mem
add   R0, 1234h         ; reg, mem
add   R0, W1            ; reg, mem
add   W1, SFR1          ; mem, reg
add   W1, R0            ; mem, reg
```

ADDB

```
addb  RL0, RH0          ; Rb, Rb
addb  RL0, [R1]         ; Rb, [Rw]
addb  RL0, [R1+]        ; Rb, [Rw+]
addb  RL0, #1           ; Rb, #data3
addb  RL0, #0ffh        ; Rb, #data8
addb  RL1, B1           ; Rb, mem
```

```
      addb  SFR1, RH0         ; mem, reg
```

ADDC

```
      addc  R0, #2            ; Rw, #data3
      addc  SFR1, #0ffffh     ; reg, #data16
      addc  R0, #0ffffh       ; reg, #data16
      addc  SFR1, 1234h       ; reg, mem
      addc  SFR1, W1          ; reg, mem
      addc  W1, SFR1          ; mem, reg
      addc  W1, R0            ; mem, reg
```

ADDCB

```
      addcb RL0, #1           ; Rb, #data3
      addcb RL0, #0ffh        ; Rb, #data8
      addcb RH7, B1           ; reg, mem
      addcb SFR1, RH0         ; mem, reg
```

9.2.2 Subtraction

There are four subtraction instructions: SUB, SUBB, SUBC, and SUBCB. SUB and SUBC are used for word (16-bit) operands. SUBC subtracts the second operand and the carry flay from the first operand. SUBC is convenient for multi-word subtraction tasks. SUBB and SUBB are the subtraction instructions used for byte (8-bit) operands. The addressing modes are listed below.

SUB

```
      sub   R0, R1            ; Rw. Rw
      sub   R0, [R1]          ; Rw, [Rw]
      sub   R0, [R1+]         ; Rw, [Rw+]
      sub   R0, #0            ; Rw, #data3
      sub   SFR1, #0ffffh     ; reg, #data16
      sub   R0, #0ffffh       ; reg, #data16
      sub   SFR1, 1234h       ; reg, mem
      sub   SFR1, W1          ; reg, mem
      sub   W1, SFR1          ; mem, reg
      sub   W1, R0            ; mem, reg
```

SUBB

```
      subb  RL0, RH0          ; Rb, Rb
      subb  RL0, [R1]         ; Rb, [Rw]
      subb  RL0, [R1+]        ; Rb, [Rw+]
      subb  RL0, #1           ; Rb, #data3
      subb  RL0, #0ffh        ; Rb, #data8
      subb  RL6, B1           ; reg, mem
      subb  SFR1, RH0         ; mem, reg
```

SUBC

```
      subc  R0, #2            ; Rw, #data3
      subc  SFR1, #0ffffh     ; reg, #data16
      subc  R0, #0ffffh       ; reg, #data16
      subc  SFR1, 1234h       ; reg, mem
      subc  SFR1, W1          ; reg, mem
      subc  W1, SFR1          ; mem, reg
      subc  W1, R0            ; mem, reg
```

SUBCB

```
      subcb RL0, #1           ; Rb, #data3
      subcb RL0, #0ffh        ; Rb, #data8
      subcb RH6, B1           ; reg, mem
      subcb SFR1, RH0         ; mem, reg
```

9.2.3 Multiplication

There are two multiplication instructions: MUL and MULU. The operands are words (16 bits). MUL is used for signed words, and MULU for unsigned words. The result of the multiplication is placed in the 32-bit MD register of the microcontroller. The addressing modes are listed below.

```
      mul   R0, R15           ; Rw
      mulu  R0, R14           ; Rw
```

9.2.4 Division

There are four division instructions: DIV, DIVL, DIVLU, and DIVU. All division operations use the 32-bit MD register of the microcontroller. The MD register holds a number which is divided by the operand. DIV and DIVU are used for word (16-bit) operands. DIV assumes that the words are signed integers. DIVU is used for unsigned integers. DIVL and DIVLU are long division instructions. MUL is used for signed words, and MULU for unsigned words. The result of the multiplication is placed in the MD register of the microcontroller. The addressing modes are listed below.

```
      div   R0                ; Rw
      divl  R0                ; Rw
      divlu R0                ; Rw
      divu  R0                ; Rw
```

9.2.5 Complement

Word or byte registers may be complemented. The complement operation, also known as one's complement, toggles each bit of the operand. It is therefore equivalent to the exclusive or of the operand with FFh or FFFFh. For example, the complement of 3Dh is C2h. Note that 3Dh+C2h=FFh

```
      cpl   R0                ; Rw
      cplb  RL0               ; Rb
```

9.2.6 Negation

Word or byte registers may be negated. The negation operation, also known as two's complement, is useful for obtaining signed integers. Two's complement is equivalent to toggling each bit of the operand and then adding one. For example, the two's complement of 3Dh is C3h. Note that 3Dh+C3h=100h, or ignoring the external carry bit, 3Dh+C3h=0. In this sense, C3 is considered as -3Dh (negative 3Dh).

```
      neg   R0                ; Rw
      negb  RL0               ; Rb
```

9.3 Logic Instructions

Logic instructions consist of AND, OR, and EXCLUSIVE-OR operations. Similar to the arithmetic operations addition and subtraction, many word- and byte-oriented addressing modes are available.

9.3.1 AND Operations

The AND instructions are used for word operands, and the ANDB, for the byte operands.

AND

```
      and   R0, R1            ; Rw. Rw
      and   R0, [R1]          ; Rw, [Rw]
      and   R0, [R1+]         ; Rw, [Rw+]
      and   R0, #0            ; Rw, #data3
      and   SFR1, #0ffffh     ; reg, #data16
      and   R0, #0ffffh       ; reg, #data16
      and   SFR1, 1234h       ; reg, mem
      and   SFR1, W1          ; reg, mem
      and   W1, SFR1          ; mem, reg
      and   W1, R0            ; mem, reg
```

ANDB

```
      andb  RL6, RL7          ; Rb, Rb
      andb  RL0, [R1]         ; Rb, [Rw]
      andb  RL0, [R1+]        ; Rb, [Rw+]
      andb  RL0, #1           ; Rb, #data3
      andb  RL0, #0ffh        ; Rb, #data8
      andb  RL2, B1           ; reg, mem
      andb  SFR1, RH0         ; mem, reg
```

9.3.2 OR Operations

The OR instructions are used for word operands, and the ORB, for the byte operands.

OR

```
      or    R0, R1            ; Rw. Rw
      or    R0, [R1]          ; Rw, [Rw]
      or    R0, [R1+]         ; Rw, [Rw+]
      or    R0, #0            ; Rw, #data3
      or    SFR1, #0ffffh     ; reg, #data16
      or    R0, #0ffffh       ; reg, #data16
      or    SFR1, 1234h       ; reg, mem
      or    SFR1, W1          ; reg, mem
      or    W1, SFR1          ; mem, reg
      or    W1, R0            ; mem, reg
```

ORB

```
      orb   RL6, RL7          ; Rb, Rb
      orb   RL0, [R1]         ; Rb, [Rw]
      orb   RL0, [R1+]        ; Rb, [Rw+]
      orb   RL0, #1           ; Rb, #data3
      orb   RL0, #0ffh        ; Rb, #data8
      orb   RH1, (B1+3)       ; reg, mem
      orb   SFR1, RH0         ; mem, reg
```

9.3.3 EXCLUSIVE OR Operations

The EXOR instructions are used for word operands, and the EXORB, for the byte operands.

EXOR

```
      xor   R0, R1            ; Rw. Rw
      xor   R0, [R1]          ; Rw, [Rw]
      xor   R0, [R1+]         ; Rw, [Rw+]
      xor   R0, #0            ; Rw, #data3
      xor   SFR1, #0ffffh     ; reg, #data16
      xor   R0, #0ffffh       ; reg, #data16
      xor   SFR1, 1234h       ; reg, mem
      xor   SFR1, W1          ; reg, mem
      xor   W1, SFR1          ; mem, reg
      xor   W1, R0            ; mem, reg
```

EXORB

```
      xorb  RL6, RL7          ; Rb, Rb
      xorb  RL0, [R1]         ; Rb, [Rw]
      xorb  RL0, [R1+]        ; Rb, [Rw+]
      xorb  RL0, #1          ; Rb, #data3
      xorb  RL0, #0ffh       ; Rb, #data8
      xorb  RL1, B1          ; reg, mem
      xorb  SFR1, RH0        ; mem, reg
```

9.4 Boolean (Bit-Oriented) Instructions

Boolean operations are used for setting and clearing individual bits. Moreover, a bit may be moved, or two bits may be compared, ANDed, ORed, or EXORed. The C166 family of microcontrollers has a powerful instruction which allows selectively replacing individual bits of a byte. Three byte operands are needed. The data byte holding the 8 bits, a mask byte indicating which of the 8 bits are to be replaced, and a third byte, giving the new values of the bits to be replaced. This operation may be viewed as two byte-oriented operations: first AND the data byte with a mask byte, clearing the bits to be replaced; then, ORing the masked data byte with the third operand which provides the new bit values. The two instructions, BFLDH and BFLDL are used for the high and low bytes of the bit-addressable word. Examples follow.

```
      bclr  BIT1              ; bit_address
      bset  BIT1              ; bit_address
      bmov  BIT1, BIT2        ; bit_address, bit_address
      bmov  BIT1, BIT2        ; bit_address, bit_address
      band  BIT1, BIT2        ; bit_address, bit_address
      bor   BIT1, BIT2        ; bit_address, bit_address
      bxor  BIT1, BIT2        ; bit_address, bit_address
      bcmp  BIT1, BIT2        ; bit_address, bit_address

      bfldh BIT_OFF1, #7, #0eh ; bit_offset, #mask8, #data8
      bfldl BIT_OFF2, #7, #0eh ; bit_offset, #mask8, #data8
```

9.5 Compare Instructions

Compare operations are similar to subtraction. The flags are modified according to the operation, except that the result of the subtraction is not kept. A conditional branching instruction typically follows a compare operation.

```
      cmp   R5, R10           ; Rw, Rw
      cmp   R6, [R3]          ; Rw, [Rw]
      cmp   R0, [R1+]         ; Rw, [Rw+]
      cmp   R0, #1            ; Rw, #data3
      cmp   SFR1, #0fedch     ; reg, #data16
      cmp   R0, #0fedch       ; reg, #data16
      cmp   SFR1, 1234h       ; reg, mem
      cmp   SFR1, W1          ; reg, mem
      cmp   R0, W1            ; reg, mem

      cmpb  RL6, RL7          ; Rb, Rb
      cmpb  RL0, [R1]         ; Rb, [Rw]
      cmpb  RL0, [R1+]        ; Rb, [Rw+]
      cmpb  RL0, #1           ; Rb, #data3
      cmpb  RL0, #0ffh        ; Rb, #data8
      cmpb  RH1, B1           ; reg, mem
      cmpb  RH0, SFR1         ; reg, mem
```

9.6 Loop Control Instructions

Loop control instructions are powerful combinations of compare and increment or decrement instructions. Word registers (Rw) are used. The register is compared to the second operand which is either a constant or the contents of a memory location. After the flags are modified according to the compare operation, the word register is either incremented or decremented. Moreover, the register may be incremented or decremented once or twice. The former is convenient for loops that progress byte-wise, and the latter, word-wise. The specific addressing modes are shown by the examples below.

```
      cmpd1 R0, #3             ; Rw, #data4
      cmpd1 R1, #1234h         ; Rw, #data16
      cmpd1 R2, W1             ; Rw, mem

      cmpd2 R3, #3             ; Rw, #data4
      cmpd2 R4, #1234h         ; Rw, #data16
      cmpd2 R5, W1             ; Rw, mem

      cmpi1 R6, #3             ; Rw, #data4
      cmpi1 R7, #1234h         ; Rw, #data16
      cmpi1 R8, W1             ; Rw, mem

      cmpi2 R9, #3             ; Rw, #data4
      cmpi2 R10, #1234h        ; Rw, #data16
      cmpi2 R11, W1            ; Rw, mem
```

9.7 Prioritization

The prioritization operation is typically used in floating-point arithmetic routines. The second operand is shifted to the left until its most significant bit is 1. The number of such shifts is stored in the first operand. Note that the number of shifts required is between 0 and 15. If the second operand is zero, the number of shifts is still set to 0, but the zero flag is set. Otherwise, the zero flag is cleared. Only word registers may be used.

```
      R0, R1                   ; Rw, Rw
```

9.8 Shift and Rotate Instructions

Shift and rotate instructions use word register operands. The number of shifts is specified either as a constant or as the contents of a word register. SHL, SHR, ROL, and ROR are the shift left, shift right, rotate left, and rotate right instructions. As the bits are shifted, zeros are introduced as new bits. The bit that is shifted out of the word is also placed into the carry flag. In the rotate operations, the bit that is shifted out is introduced as the new bit. That is, the rotate instructions are over 16 bits. The bit that is shifted out and introduced as the new bit in a rotate instruction is also copied into the carry flag. The arithmetic shift right (ASHR) operation preserves the sign of a signed integer operand by introducing either zeros or ones, depending on the most significant bit of the operand.

SHIFT

```
      shl   R0, R1            ; Rw, Rw
      shl   R1, #1111b        ; Rw, #data4
      shr   R0, R1            ; Rw, Rw
      shr   R1, #1111b        ; Rw, #data4

      ashr  R0, R1            ; Rw, Rw
      ashr  R1, #1111b        ; Rw, #data4
```

ROTATE

```
      rol   R0, R1            ; Rw, Rw
      rol   R1, #1111b        ; Rw, #data4
      ror   R0, R1            ; Rw, Rw
      ror   R1, #1111b        ; Rw, #data4
```

9.9 Data Transfer Instructions

There are four types of move instructions: MOV for word operands, MOVB for byte operands, MOVBS for signed extended moves of bytes into words, and MOVBZ for zero-extended moves of bytes into words. The latter two are used when a byte is to be moved into a word. In either MOVBS or MOVBZ, the bytes are moved into the low byte of the destination. The high byte of the destination is set to zero in MOVBZ, used when the source byte is an unsigned integer. MOVBS fills the high byte of the destination with either ones or zeros, depending on the most significant bit of the source byte. MOVBS is used when the source byte is a signed integer.

In register indirect addressing, that is, when a register holds the address of the source or destination, the register may be post incremented or pre-decremented. In byte moves, the register value is changed by one, and in word move operations, by two. For instance,

```
            mov   R0, [R1+]
```

moves the word whose address is in register R1 into R0. Then, R1 is set to R1+2. Similarly,

```
            movb  [-R0], RH1
```

moves a byte using the indirect addressing mode. But first, the value of R0 is set to R0-1. Then, the byte in RH1 is moved to the address specified by R0.

MOV

```
      mov   R0, R1             ; Rw, Rw
      mov   R0, #15            ; Rw, #data4
      mov   SFR1, #65535       ; reg, #data16
      mov   R0,  #65535        ; reg, #data16
      mov   R0, [R1]           ; Rw, [Rw]
      mov   R0, [R1+]          ; Rw, [Rw+]
      mov   [R0], R1           ; [Rw], Rw
      mov   [-R0], R1          ; [-Rw], Rw
      mov   [R0], [R1]         ; [Rw], [Rw]
      mov   [R0+], [R1]        ; [Rw+], [Rw]
      mov   [R0], [R1+]        ; [Rw], [Rw+]
      mov   R0, [R1 + #1234h]  ; Rw, [Rw + #data16]
      mov   [R0 + #1234h], R1  ; [Rw + #data16], Rw
      mov   [R1], W1           ; [Rw], mem
      mov   W1, [R1]           ; mem, [Rw]
      mov   W1, SFR1           ; mem, reg
      mov   SFR1, W1           ; reg, mem
```

MOVB

```
      movb  RL0, RH1           ; Rb, Rb
      movb  RH0, #15           ; Rw, #data4
      movb  RH0, #16           ; reg, #data8
      movb  RL7, [R1]          ; Rb, [Rw]
      movb  RH0, [R1+]         ; Rb, [Rw+]
      movb  [R0], RL1          ; [Rw], Rb
      movb  [-R0], RH1         ; [-Rw], Rb
      movb  [R0], [R1]         ; [Rw], [Rw]
      movb  [R0+], [R1]        ; [Rw+], [Rw]
      movb  [R0], [R1+]        ; [Rw], [Rw+]
      movb  RL0, [R1 + #1234h] ; Rb, [Rw + #data16]
      movb  [R0 + #1234h], RH1 ; [Rw + #data16], Rb
      movb  [R4], W1           ; [Rw], mem
      movb  W1, [R4]           ; mem, [Rw]
      movb  W1, SFR1           ; mem, reg
      movb  SFR1, W1           ; reg, mem
```

MOVBS

```
      movbs R0, RL1            ; Rw, Rb
      movbs W1, SFR1           ; mem, reg
      movbs SFR1, W1           ; reg, mem
```

MOVBZ

```
      movbz R0, RL1            ; Rw, Rb
      movbz W1, SFR1           ; mem, reg
      movbz SFR1, W1           ; reg, mem
```

9.10 Branching Transfer Instructions

There are four types of branching instructions; jumps, calls, traps (interrupts), and returns. Each type consists of one or more specific forms. The conditional branching instructions use a common set of condition codes.

Condition Code	**Description**
cc_UC	unconditional
cc_Z	zero
cc_NZ	not zero
cc_V	overflow
cc_NV	no overflow
cc_N	negative
cc_NN	not negative
cc_C	carry
cc_NC	no carry
cc_EQ	equal
cc_NE	not equal
cc_ULT	unsigned less than
cc_ULE	unsigned less than or equal
cc_UGE	unsigned greater than or equal
cc_UGT	unsigned greater than
cc_SLT	signed less than
cc_SLE	signed less than or equal
cc_SGE	signed greater than or equal
cc_SGT	signed greater than
cc_NET	not equal and not end of table

Jump instructions consist of absolute (JMPA), indirect (JMPI), relative (JMPR), and inter-segment (JMPS) jumps. Jumps may also be conditioned on the state of a bit: if bit set (JB) and if bit not set (JNB). There are two additional bit-dependent jumps: JBC and JNBS. These are similar to JB and JNB, except that, after the instruction, JBC clears the bit, and JNBS sets the bit.

```
      jmpa  cc_UC, moves      ; cc, code_address
      jmpi  cc_UC, [R0]       ; cc, [Rw]
      jmpr  cc_UC, moves      ; cc, relative_address
      jmps  1, 0              ; segment, code_address

      jb    BIT1, moves       ; bit_address, relative_address
      jbc   BIT1, moves       ; bit_address, relative_address
```

```
jnb   BIT1, moves       ; bit_address, relative_address
jnbs  BIT1, moves       ; bit_address, relative_address
```

The four addressing modes of the jump instructions are also valid for the call instructions: CALLA, CALLI, CALLR, and CALLS.

```
calla cc_UC, moves      ; cc, code_address
calli cc_UC, [R0]       ; cc, [Rw]
callr moves             ; cc, relative_address
calls 1, 0              ; segment, code_address
```

In addition, an unconditional call instruction, PCALL pushes the first operand onto stack and makes a call to the second operand

```
pcall SFR1, moves       ; reg, code_address
```

All interrupts may be invoked by software by the TRAP instruction. The single operand specifies the trap (interrupt) number. The trap instruction pushes the Code Segment Pointer (CSP), the Instruction Pointer (IP), and the Program Status Word (PSW) before the branch.

```
trap  #1ch              ; trap #7
```

The instruction RET pops the Instruction Pointer (IP) to return from a subroutine. RET is typically used at the end of a subroutine which was branched to using a CALLA , CALLI, or CALLR instruction. RETS additionally pops the Code Segment Pointer (CSP) from stack. RETS is typically used at the end of a subroutine which was branched to by a CALLS instruction. RETP is similar to the RET instruction, except that a word is poped off stack into the operand. RETP is useful when the subroutine is to return an integer (word) value. Finally, RETI pops PSW, IP, and CSP from the stack. It also resets the microcontrollers interrupt system to indicate the end of an interrupt (trap). RETI is typically used at the end of interrupt service routines.

```
ret
rets
retp  SFR1              ; reg
reti
```

9.11 System Stack Instructions

System stack instructions are the push, pop and the switch context (SCXT) instructions. Only words may be pushed or popped. If byte values are to be stored on stack, they must first be placed into words. For example, although RL0 cannot be pushed, R0 can be pushed. PUSH places the word operand on stack, and POP removes them, transferring the top of stack to the operand. SCXT is a combination of push and move operations. The register specified by the first operand is pushed. Then the second operand is moved into the register. The second operand may be a constant or the contents of a memory location.

```
pop   SFR1              ; reg
pop   R1                ; reg
push  SFR1              ; reg
push  R1                ; reg
scxt  SFR1, #1234h      ; reg, #data16
scxt  R1, #1234h        ; reg, #data16
scxt  SFR1, W1          ; reg, mem
scxt  R1, W1            ; reg, mem
```

Note that SCXT is a powerful instruction when used with the Context Pointer (CP). For example,

```
      scxt  CP, #0FC80h       ; select a new register bank
      .
      .
      .
      .
      pop   CP                ; re-institute the old register bank
```

pushes CP on stack and modifies it to be FC80h. This determines a new address of the general purpose registers (GPRs). Once the section of code is completed, the old register bank may be re-instituted by simply popping CP.

9.12 System Control Instructions

The system control instructions do not take any operands. They include software reset (SRST), idle mode (IDLE), power down (PWRDN), service watchdog timer (SRVWDT), disable watchdog timer (DISWDT), and end of initialization (EINIT).

System control instructions are protected instructions that take four bytes. The first byte of the instruction is repeated twice again as instruction bytes 3 and 4. If the processor fails to receive these two duplicate bytes, an interrupt (the protection fault trap) is invoked.

```
      srst
      idle
      pwrdn
      srvwdt
      diswdt
      einit
```

The 167 has additional instructions to access the extended features and the extended register set. These instructions are listed below.

```
      atomic #3               ; #data2
      extr   #3               ; #data2
      extp   R0, #3           ; Rw, #data2
      extp   #10b, #3         ; #page10, #data2
      extpr  R0, #3           ; Rw, #data2
      extpr  #01b, #3         ; #page10, #data2
      exts   R0, #3           ; Rw, #data2
      exts   #11b, #3         ; #seg8, #data2
      extsr  R0, #3           ; Rw, #data2
      extsr  #00b, #3         ; #seg8, #data2
```

9.13 Miscellaneous Instructions

The no operation instruction (NOP) simply wastes a instruction cycle. It takes no operands.

```
      nop
```

10.0 Rc66 -- THE READS166 C COMPILER

10.1 Native 32-Bit Compiler

Rc66 is designed as to be an integral component of the READS166 Integrated Development Environment, rather than a standard ANSI C compiler. The rapid application development aspects of READS166 have been the guiding theme in designing Rc66.

The READS166 compiler, the Rc66 is implemented in native 32-bit code as a DLL. It is specifically written for the 32-bit MS Windows (TM) environments Windows95 and WindowsNT. As such it takes advantage of the Windows graphical user interface, memory and file management systems.

10.2 One-Step Compilation

Rc66 produces HEX code from C source in one step, thus eliminating the need for linking. The speed and capabilities of contemporary personal computers are not pushed even if all source code is recompiled each time there is a modification. This one-step compilation approach allows reusable code to be stored in source form rather than in object form in libraries. Library management is simplified. The code is simply recompiled for different versions, for example for debug versus release versions, or for different memory maps.

Rc66 has an internal profiler that by default will not produce machine code for functions not called in the program. This way your final HEX code will not grow even if you include source code from an archive that contains unwanted functions. You may override this default behavior by simply clicking on a check box in the C options dialog.

10.3 Special Function Registers

Rc66 handles the Special Function Registers (SFRs) and their bits in a somewhat different fashion than similar C compilers for microcontrollers. Rather than introducing C language extensions such as SFR, SFRBIT, XSFR, etc, Rc66 introduces the storage class specifier “predef” which simply instructs the compiler to refer to the variable by its name rather than allocating memory for it. The keyword is short for “predefined.” This approach is similar to the use of the storage class specifier “register” where the C variable is kept in a register. With this approach, standard C variable type specifiers and type qualifiers may be used. For example,

```
predef unsigned int SYSCON;
```

instructs the compiler that SYSCON is an unsigned integer which when generating code, should be referred to by its name. Predefined variables must be declared as global variables.

Rc66 uses the type “bit” only for predefined variables, for example

```
predef bit IEN;
```

In the following example, the value of the SYSCON register of the 166 processor is modified to run at zero wait states.

```
// --- Uses No Wait States ---
predef unsigned int SYSCON;

void main(void){

 SYSCON &= 0xF;
    .
    .
    .
}
```

```
// --- End Of Program ---
```

10.4 Interrupt Service Routines

Rc66 fully supports interrupts. Unlike similar compilers, Rc66 does not force binding the Interrupt Service Routine (ISR) to the interrupt source in compile time. Rather, the ISR address is placed into the interrupt vector of the microcontroller along with an absolute jump instruction. This way, ISRs may be switched in run time.

10.5 Writing Interrupt Service Routines (ISRs)

An interrupt-driven approach requires the following steps.:

1. Writing the interrupt Service Routine (ISR)
2. Configuring the Interrupt Source
3. Setting the interrupt vector and interrupt priority
4. Enabling the interrupt
5. Optionally, initiating the process
6. Cleaning up after the process is completed.

These steps are illustrated with an example given in the on-line help section Software Experiments, and in Appendix E. The example uses port P2.8 as an external interrupt input. When the state of P2.8 makes a one-to-zero (high-to-low) transition, an interrupt is generated.

1. Writing the ISR

Since interrupt service routines are called upon hardware signals, they cannot have arguments or return a value. Consequently, any information exchange between the foreground code and interrupt service routines must be accomplished by global variables that are visible to all parts of the program. In the example given below, the interrupt service routine uses the global variable nCount to pass information to the foreground program.

The keyword "interrupt" instructs the compiler that the function must be written as an interrupt service routine. This essentially means pushing and popping all relevant registers and returning with an RETI instruction. Rc66 also uses a different expression stack set up mechanism for interrupt service routines. The "interrupt" keyword should be considered similar to the type qualifier keyword "volatile."

We use CAPCOM8 to generate interrupts on the falling edge of the input. Thus the lowest nibble of CCM2 is set to 2. Each time P2.8 makes a one-to-zero transition an interrupt will be generated.

The ISR is given below.

```
/* -------------------- */
void interrupt Isr(void){
 nCount++;  // increment count
}
```

The support function SetIntVec() is placed in the archive UTILITIES in module INTUTIL.C. You may use this function in your applications, or remove the debug suppression directive and single step through SetIntVec() to inspect its operation.

2. Configuring the Interrupt Source

Most interrupts involve some type of peripheral device that needs to be configured. In this example, the capture-compare unit is responsible for generating the interrupt. The last nibble of SFR CCM2 is set to 2 to instruct the capture-compare unit to generate interrupts on the falling edge of the input P2.8.

```
CCM2&=0xFFF0;
CCM2|=2;        // CCMOD8=2 (interrupt on neg edge)
```

3. Setting the Interrupt Vector and Priority

The 166 family of microcontrollers have dedicated interrupt vectors for the various hardware interrupts. A jump instruction to the interrupt service routine must be placed at these predefined vectors. The archive function SetIntVec() sets the interrupt vector. SetIntVec() takes two arguments: the vector and the address of the interrupt service routine.

The CAPCOM interrupt control register CC8IC is initialized to 4Ch, which sets its enable flag and assigns it an interrupt priority of 3 with group level 0.

```
    // --- set interrupt vector ---
    SetIntVec(0x60, Isr);

    // --- set interrupt priority ---
    CC8IC=0x4C;     // enable CAPCOM8 interrupts - low priority
```

4. Enabling the interrupt

Although we enabled the CAPCOM0 interrupt, the CPU will not process interrupts unless the master interrupt enable flag IEN located in the PSW register is set. Simply set IEN=1 to allow interrupts.

```
    // --- turn master interrupt enable flag on ---
    IEN=1;          // turn master interrupt enable on
```

5. Initiating the Process

Some interrupts are generated at the end of a process. For example, a timer interrupt is generated when the timer rolls over. The timer must be started to initiate the process. Similarly, an analog-to-digital conversion completion interrupt needs the ADC to be started. In this example, the external interrupt is not a process completion interrupt, and thus this step is skipped.

6. Cleaning up after the process is completed

Upon termination, CAPCOM8 interrupts are disabled by clearing the CC8IE flag and giving it priority 0. This is accomplished by setting CC8IC to 0. Similarly, the master interrupt enable flag IEN is cleared.

```
    DP2_9=0;        // P2.9 is an input
    IEN=0;          // turn master interrupt enable off
    CC8IC=0;        // disable CAPCOM8 interrupts
```

The entire program may be found in the executable project PollInt on the distribution disk. Further information is also available in the on-line help section Software Experiments'.

10.6 Mixing C with Assembly Code

One of the aspects in which embedded code differs from general-purpose computing code is that it often combines high level code with low level machine code or assembly code. The READS166 IDE and Rc66 strives to make mixed language programming as painless as possible.

READS166 allows projects to freely combine C modules with assembly modules. An assembly module will most likely contain functions written in assembly language. These functions are then called from the C code. Rc66 allows passing an integer argument to the assembly routine, and returning an integer argument from the assembly routine. The first register in the used registers set is reserved for this purpose. Rc66 defines the macro ASMREG as the current register reserved for such passing to make your code more portable.

The approach is illustrated by an example that uses an assembly language subroutine to give the successor of an integer. The process involves the following steps.

1. Write your assembly function

Save your assembly language subroutine in an assembly module. In this example, consider the follow subroutine.

```
      ; --- assembly language subroutine ---
      Successor:
            add   ASMREG, #1
            ret
      ; --- end of assembly language subroutine ---
```

2. Write your C program

Your C program will call the assembly routine. The C program needs to have type information about the function it is calling. Thus a function prototype of the assembly routine is necessary. Consider the following code.

```
      // --- function prototype ---
      int Successor(int);   // int-int

      // --- main code ---
      main(void){
      int n, k;

       for(n=0; n<10; n++)
        {
         k=Successor(n);
         }
      }
      // --- end of C code
```

Place your C code in a C module under the project.

3. Build the project

When built, Rc66 will combine the C code and the assembly code.

Notes

1. You are not restricted to write subroutines in assembly code, nor are you restricted to call all assembly routines from the C code. Assembly code may consist of many routines, only a few of which are made visible to the C code. Similarly, you may use all Ra66 constructs, such as ORG (origin), or DB (define byte) to place data structures or code in memory. For example, you may prefer writing your interrupt service routines in assembly and placing jumps at the interrupt vectors.

2. For further examples of mixed-language programming refer to the executable project MIXED and MIXEDFACTORIAL.

10.7 The Debugger

Debug functions are included in Rc66 which generates instructions embedded into the final HEX code. The debugger undertakes background communications with a target Rigel board to single step, animate, set break points, and watch variable values.

Debug instructions increase the size of your final code and marginally slow down its execution even when no break points are set. You may switch the debug mode on or off from a check box in the Project Build Options dialog. You may also selectively suppress the generation of debug instructions for individual functions using the "#pragma DebugOff" directive. This way, while keeping the debug option enabled, you may suppress debug instructions being generated for functions you have already debugged. For example, modules placed in archives would typically suppress debug code generation.

11.0 A BRIEF REVIEW OF C

11.1 C Language Philosophy

C is by far the High-Level Language (HLL) of choice. C is the first truly portable computer language. There is a C compiler for virtually all processors. Moreover, C will most likely continue be the dominant HLL for future generations of processors. This means you may port your code to future hardware with ease. C, being closer to assembly language, makes it a good language for microcontrollers. Compared to other HLLs, such as BASIC, you can have finer control over the microcontroller hardware with C.

C forces the code to be more structured. It is not uncommon to see unstructured code with many forward and backward jumps (among programmers this is referred to as spaghetti code) in assembly or BASIC. C imposes structure by minimizing or eliminating labels, and by forcing variable declarations.

C is a highly capable language when it comes to making use of previously compiled code. Traditionally libraries of precompiled code would be linked with C code to produce final executable code. Thus, making use of external components (external functions or variables, for example) is fundamental to the success of C. With the appropriate libraries, C may be customized to undertake demanding tasks it was not originally intended to do. For example, with a good complex number library, C may be used as a number-crunching platform. This chameleon-like feature of C makes it the language of choice in scientific computing as well as writing large-scale applications such as computer graphics, word processing, desktop publishing, data base management, communications programming, and networking.

C is not a language without its critics. The language was designed for writing operating systems. Numerical work were not top priority issues in designing C. For example, the ANSI standard only requires trigonometric functions to be provided in double-precision versions, although many compilers, do provide them in single-precision as well. Similarly, handling multi-dimensional array pointers may seem difficult to the new comer.

Many programmers, when first introduced to C complain that it is a very cryptic language. Granted, it is easier to write opaque code in C than it is in, say BASIC. Cryptic code usually is a result of trying to shorten the code. It almost always results in reducing code readability. Although it is possible to write cryptic code in C it is not necessary.

Finally, C imposes fewer restrictions on the programmer. For example, it is not a strict type checking or strict range checking language. This gives the programmer more freedom and power, at the expense of added responsibility to write good crash-proof code. But this is hardly new to assembly programmers. In fact it is this freedom that makes C a convenient HLL for microcontrollers.

11.1.1 Ingredients of a C Program

A C program consists of functions, variables, and statements. These functions may be user provided, or may come from one or more Run-Time Libraries (RTLs). A RTL is a collection of precompiled functions that are linked to your program to produce the final executable code.

Sometimes C is called a function-oriented language. All C instructions must belong to a function. In fact the entire program is initiated when a special function called “main” is called. When main returns, your program terminates. The latter must be reviewed in the case of embedded controller code, since embedded controller code may be required never to terminate.

The traditional “Hello World” program below shows some of the ingredients of the language.

```
#include <stdio.h>
void main(void){
    printf("\nHello World\n");
}
```

The void preceding ``main'' indicates that function main does not return a value. Similarly, the keyword “void” which appears inside the set of parentheses immediately following “main” specifies that the function “main” has no arguments. That is, no parameters are passed to the function.

C string constants are written between double quotation marks. The characters ``\n'' prints a ``new line'' character, which brings the cursor onto the next line.

11.1.2 Code Appearance and Style

The code starts with a series of comments indicating its purpose, as well as its author. It is considered good programming style to identify and document your work (although, sadly, most people only do this as an afterthought). Comments can be written anywhere in the code: any characters between /* and */ are ignored by the compiler and can be used to make the code easier to understand. The use of variable names that are meaningful within the context of the problem is also a good idea.

11.2 Functions

Function Prototypes

Functions are declared by specifying the type and number of arguments they take and by the type of value they return. Such declarations are called function prototypes. Consider, for example, the prototype of a successor function which takes an integer and returns the next integer:

```
int GetNextInteger(int);
```

Semicolons are used as delimiters to mark the end of the statements. Blocks of statements are put in curly brackets (also referred to as braces). A collection of statements placed in curly brackets is called a compound statement, which acts as a statement.

All C statements are defined in free format, i.e., with no specified layout or column assignment. (Old FORTRAN programmers will remember the significance of column 6 and 7!) Whitespaces (tabs or spaces) are never significant, with the exception of being a part of a character string. Thus it is possible to write the “Hello World” program as follows

```
#include <stdio.h>void main(void){printf("\nHello World\n");}
```

which sometimes leads to a cryptic appearance.

11.3 Variables

11.3.1 Scalars

Variable names are arbitrary (with some compiler-defined maximum length, typically 32 characters). C uses the following standard variable types:

int	integer variable
short	short integer
long	long integer
float	single precision real (floating point) variable
double	double precision real (floating point) variable
char	character variable (single byte)

C requires the variables to be defined before they are used. The following example illustrates the use of variables.

```
main(void){
int nNumber, nSuccessor;

 nNumber=1;
 nSuccessor=GetNextNumber(nNumber);
}

int GetNextInteger(int n){
 return n+1;
}
```

C is case sensitive, so function and variable names must be case consistent throughout your program. For example, nNumber and nNUMBER are not the same!.

In this example, variables are defined within the compound statement. Such variables are called local variables. They may be used only within the compound statement in which they are defined. All local variables must be defined before any other statements.

Alternatively, you may have global variables, defined outside the compound statements. These are called global variables. For example,

```
int nNumber, nSuccessor;

main(void){

 nNumber=1;
 nSuccessor=GetNextNumber(nNumber);
}

int GetNextInteger(int n){
 return n+1;
}
```

defines the two integers nNumber and nSuccessor as global variables.

In strict C, global variables may only be used in compound statements that appear below their definitions. Rc66 does not impose this limitation.

Variables may be initialized when defined. Assembly programmers will recognize the similarity between these definitions and the DB pseudo operation.

```
int n=0;
```

not only defines the integer n, but it also sets its initial value to 0.

11.3.2 Pointers

Similar to the BASIC peek and poke functions, C allows direct access to memory. In fact, C provides a very powerful method of memory access, which makes it the language of choice to write memory intensive applications.

The approach is based on storing the memory address as a variable. Such a variable is called a pointer (to memory). Pointers variables (variables which store memory addresses) are declared using the asterisk. Below, we define an integer n and a pointer to an integer pn.

```
int n, *pn;
```

You may extract the memory address of a given variable by the C operator ‘&’. Thus, the statement

```
pn=&n;
```

gets the memory address of the integer n and places it into the pointer variable pn. The ampersand operator is referred to the reference operator.

The opposite operation is also needed. The contents of the memory referenced by a pointer is obtained using the ``*" operator, referred to as the dereference operator. Provided that pn contains the memory address of the variable n, *pn has the same value as n. For example,

```
*pn=5;
```

is equivalent to

```
n=5;
```

11.3.3 Arrays

Arrays of any type can be formed in C. The syntax is simple:

```
type name[dim];
```

For example,

```
int nADC[16];
```

defines an array of 16 integers. C arrays start at position 0. The elements of the array occupy adjacent locations in memory. C treats the name of the array as if it were a pointer to the first element. This is important in understanding how to do arithmetic with arrays. Thus, if v is an array, *v is the same as v[0], *(v+1) is the same as v[1]:

11.4 Constants

11.4.1 Compiler Directives

You can define constants of any type by using the #define compiler directive. Its syntax is simple--for instance

```
#define ANGLE_MIN 0
#define ANGLE_MAX 360
```

would define ANGLE_MIN and ANGLE_MAX to the values 0 and 360, respectively. C distinguishes between lowercase and uppercase letters in variable names. It is customary to use capital letters in defining global constants.

11.5 Statements

C has six basic classes of statements:

Compound Statements
Expressions
Iteration Statements
Selection Statements
Jump Statements
Labeled Statements

Expressions are the basic staple of any programming language. Statements are usually built around one or more expressions.

11.5.1 Compound Statements

Compound statements collect a set of statements as well as definitions of local variables. Compound statements play a central role in iteration statements or selection statements when more than one statement needs to be executed during an iteration, or as a result of a condition. Consider, for example, the iteration statement

```
while(expression) statement
```

In most cases, the statement of the above while loop needs to perform several tasks. This is easily accomplished by a compound statement. In effect, a compound statement introduces a set of statements which, from a syntactic point of view, act as a single statement.

```
while(expression)
{
 statement_1;
 statement_2;
 .
 .
 .
 statement_n;
 }
```

11.5.2 Expressions

Expressions are the basic staple of any programming language. Perhaps the most commonly used expression is the assignment expression, such as

```
n=5;
```

C allows many assignment operators besides the simple equal assignment.

=	assignment
+=	addition assignment
-=	subtraction assignment
*=	multiplication assignment
/=	division assignment
%=	remainder/modulus assignment
&=	bitwise AND assignment
|=	bitwise OR assignment
^=	bitwise exclusive OR assignment
>=	right shift assignment

The format “variable operation=” is short for “variable=variable operation”. For example,

```
n+=5;
```

is equivalent to

```
n=n+5;
```

C allows you to put multiple expression in the same statement, separated by a comma. The expressions are evaluated in left-to-right order. The value of the overall expression is then equal to that of the rightmost expression.

For example,

```
n=((k=1),2);
```

is equivalent to the two assignments

```
n=2;
k=1;
```

Similarly, when used as a function argument,

```
f(n,(k=1,k+1),1);
```

is equivalent to the assignment and function call

```
k=1;
f(n,2,1);
```

The comma operator is useful in some cases, such as in iteration statements, but in general, overusing the comma operator produces unreadable code.

C conditions are also expressions. If an expression is evaluated to be zero, the condition is considered to be false. Otherwise the condition is true.

11.5.3 Conditions

C conditions are also expressions. If an expression is evaluated to be zero, the condition is considered to be false. Otherwise the condition is true.

C provides many conditional or logical operations to simplify the evaluation of expressions to be used as conditions.

==	equal to
!=	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
&&	logical and
||	logical or
!	logical not

For example, the expression

```
(j==2)
```

has the value 1 only if j is equal to 2.

11.5.4 Iteration Statements

Iteration statements provide code loops which are structured ways to accomplish repetitive algorithmic procedures. C supports three basic types of iteration statements: the while statement, the do-while statement, and the for statement. The syntax of each type of iteration statement is given below.

```
while(expression) statement
do statement while (expression);
for(expression;expression;expression) statement
```

Note that the statements may be compound statements, possibly (and often) containing other iteration statements. The while statement evaluates its expression. The statement is executed if the expression is nonzero. The process is repeated until the expression is evaluated to be zero. For example,

```
void main(void){
int n=0;

SendStr(“Hello World\n”);
while(n<10)
 {
  SendStr(“hello again\n”);
```

```
          n++;
          }
        }
```

prints “Hello World” followed by ten lines of “hello again.” Note that the statement of the while statement is a compound statement. This way, the statement accomplishes more than one task: it prints a string, and it increments n. The latter task is most important, since otherwise the while expression would never be evaluated as zero, hence resulting in an endless loop.

Endless loops are not all evil though. Neither is the statement always necessary. Consider for example the while statement

```
        .
        .
        .
        while(P2_0);
        .
        .
        .
```

where P2_0 is the value of port 2.0. The program will remain at the while loop until the state of the port bit becomes 0. Note that the program simply waits (or hangs) at the while statement without executing any other statement.

The do-while statement is similar to the while statement, except that the statement is first executed, and the expression evaluated afterwards. The above example could be rewritten as,

```
        void main(void){
        int n=0;

        SendStr(“Hello World\n”);
        do
         {
          SendStr(“hello again\n”);
          n++;
          } while(n<10);
        }
```

Perhaps the C for statement is the most often used iteration statements by programmers new to C. This statement closely resembles the BASIC for statement and the FORTRAN do statement. There are three expressions in the C for statement: the initialization expression, the condition expression, and the iteration expression.

```
        for (initialization_expression; condition_expression; iteration_expression) statement
```

The for statement may be viewed as a special case of the C while statement, equivalent to the following:

```
  {
   initialization_expression;

   while (condition_expression)
     {
      statement;
      iteration_expression;
      }
   }
```

The above example is now written with the for statement.

```
void main(void){
int n;

SendStr(“Hello World\n”);
for(n=0; n<10; n++) SendStr(“hello again\n”);
}
```

Note that the initialization of the iteration counter n is now moved to the for statement. This is not necessary, however, since any one of the for expressions may actually be null expressions. That is, the following code has the same effect.

```
void main(void){
int n=0;

SendStr(“Hello World\n”);
for( ; n<10; n++) SendStr(“hello again\n”);
}
```

It was mentioned that infinite loops may have their use in programming. In addition, C provides a good mechanism to break out of a loop. The two C keywords “continue” and “break” provide this additional control. The “continue” command skips the rest of the statements and repeats the iteration. The “break” command terminates the iteration and exits from the loop. Again, consider our example.

```
void main(void){
int n=0;

SendStr(“Hello World\n”);
for( ; ; n++)
 {
  SendStr(“hello again\n”);
  if(n>=9) break;
  }
}
```

Here, we have made two changes. First we replaced the old statement with a compound statement. Next, we removed the condition from the for statement. The loop is now terminated when n reaches 9 by the break command. Note that the iteration limit is 9 since n will go from 0 to 9 and hence print the string 10 times.

As an extreme case, consider

```
void main(void){
int n=0;

SendStr(“Hello World\n”);
for( ; ; )
 {
  SendStr(“hello again\n”);
  if(n>=9) break;
  n++;
  }
}
```

Although such programming style may at first seem unusual, it is actually practiced by some. Similarly, it is perfectly legitimate to write

```
void main(void){
int n;

SendStr(“Hello World\n”);
for(n=0; n<10; n++, SendStr(“hello again\n”);
}
```

moving the statement into the for expression. The programmer should strive not only for correct code but for readable code. With attention to variable and function names as well as programming style as illustrated by these examples, C could become quite a self-documenting programming language.

11.5.5 Selection Statements

There are two types of selection statements in C: the if (and if-else) statement, and the switch statement.

The if and if-else statements have a straightforward structure:

```
if(expression) statement
if(expression) statement else statement
```

For example, consider

```
void main(void){
int n;

for(n=0; n<10; n++)
  if(n%2) SendStr(“odd\n”);
   else SendStr(“Even\n”);
}
```

This example prints a series of strings (Even, Odd, …). Note that although the syntax of the program is correct, many programmers prefer to place any statement following an if(expression) inside curly brackets, as below.

```
void main(void){
int n;

for(n=0; n<10; n++)
 {
  if(n%2) SendStr(“odd\n”);
   else SendStr(“Even\n”);
  }
}
```

This improves readability by clearly isolating the statement to be executed when the expression is nonzero.

The switch statement is a powerful construct with the following syntax:

```
switch (expression)
 {
  case const_expression_1: statement
  case const_expression_2: statement
      .
      .
```

```
    default: statement
    }
```

The expression must evaluate to an integral value. The value is compared to each constant expression. If an equal constant expression is found, the corresponding statement is executed. Note that the cases are actually labels. The program will normally continue executing after the statement. Thus you will frequently find switch statements in the form

```
switch (expression)
   {
    case const_expression_1: statement;
                             break;
    case const_expression_2: statement
                             break;
        .
        .
    default: statement
    }
```

11.6   Comments

C comments start with the character pair ‘/*’ and terminate with the pair ’*/’. For example,

```
/*
the traditional Hello World program
another line of comments
and yet another
*/

/* --- header files --- */
#include <stdio.h>

/* --- main function --- */

void main(void){
    printf("\nHello World\n"); /* print the string */
}
/* --- end of code --- */
```

illustrates the use of C comments.

Assembly language programmers may find writing 4 extra characters per comment a bit too much, since anything from a semicolon to the end of the line is a comment in assembly language. C++ introduced a similar type of comments where a double forward slash denotes the beginning of the comment. As in assembly language, the comment terminates at the end of the line. Although strict C compilers will not recognize such comments, Rc66 does. It is then possible to write

```
/*
 the traditional Hello World program
 another line of comments
 and yet another
*/

// --- main function ---
```

```
void main(void){
    SendStr("\nHello World\n"); // print the string
}
// --- end of code ---
```

Note that the C-type comments are still convenient for multi-line comments.

11.7 Standard (Run Time) Libraries

You will notice that the central role is played by the function “printf” (short for print function) which is actually a library function, rather than a built in C feature. That is, somebody has written the function “printf().” The first line is a compiler directive instructing the compiler to include the file “stdio.h” in which a prototype of the function “printf” may be found. The file “stdio.h” is called a header file (thus the extension ‘h’.)The compiler must also be instructed to link the code with the standard libraries containing the precompiled version of “printf.” Unlike other HLLs, to include a header file or to link with the proper library is the responsibility of the programmer. RTL functions such as “printf” are now standard in ANSI C. The K & R textbook lists the content of these and other standard libraries in its appendix.

Rc66 is not strictly an ANSI C compiler. It is written with a graphical Integrated Development System (IDE) in mind. Rc66 does not require function prototypes. Rather, it performs a scan pass over the code to see which functions are used, and which functions are available. Thus, in Rc66 the “Hello World” program becomes

```
void main(void){
    SendStr("\nHello World\n");
}
```

Note that the function SendStr() accomplishes the same as “printf,” that is, prints the given string. It is a part of serial communications routines. SendStr() actually sends the characters out the serial port of the microcontroller.

11.8 References

An excellent textbook on C by two well-known and widely respected authors is:
The C Programming Language -- ANSI C Brian W. C. Kernighan & Dennis M. Ritchie, Prentice Hall, 1988

12.0 Rc66 SOFTWARE/HARDWARE EXPERIMENTS

12.1 The Hardware

The RIO-MP (also referred to as RIO or the RIO board) is a general purpose I/O board designed for use with Rigel Corporations' family of 8 and 16-bit embedded controllers. The development and implementation of application-specific microcontroller based prototype circuits is significantly simplified with the RIO breadboard area, the three terminal strips, and the user input/output devices (UIOD). These provide flexibility for connecting prototyping components to the microcontroller lines, and for developing and debugging user-designed analog and digital application circuits.

The RIO board in conjunction with the READS166 debugging features allows developing and interactively debugging user code. Consider, for example, a software loop that repeatedly reads an ADC channel and breaks out of the loop only when the value reaches a predefined threshold. By actually connecting the microcontroller ADC input to a potentiometer and setting a break point immediately after the loop, one can debug the hardware/software system much like it will appear in the final application. The program will run in READS166 and reach the breakpoint and stop only after one turns the potentiometer past the threshold value.

12.2 The Software

The READS166 C compiler has the capability to generate additional code necessary for interrupt service routines. This allows testing software written in an interrupt-driven architecture. The READS166 debugger will step through interrupt service routines once external (or internal) signals are received to trigger the interrupts.

In the following sections, we present sample code and hook up instructions. Such hook up is limited to connecting the various microcontroller ports to UIODs. When appropriate, ribbon cables are used to connect several signals to ports. For example, a four-wire ribbon cable is used to connect the LCD data bus to Port 2.0-P2.3. This further simplifies hook up.

12.3 Simple Input/Output Experiments

These experiments illustrate simple bit-wise or word-wise inputs and outputs.

Using Single-Bit Inputs and Outputs
Polled Analog-to-Digital Conversion
Simple LCD Operation
Combining Modules: ADC with LCD
Stepper Motor Control

12.3.1 Using Single-Bit Inputs and Outputs

This is a simple example illustrating the use of the ports as simple input/outputs. The example also shows how the simple serial communications routines may be used to display strings in the TTY window.

Hardware connections

Port P2.8 is used as an input and P2.9 as an output. Connect push button PB0 of the RIO board to P2.8 of the microcontroller. Also connect P2.9 to LED0 of the RIO board.

Software

Load the project PIOPOLL. PIOPOLL consists of a main module and the serial communications utility module S0.C from the archive project UTILITIES.

This experiment uses port bits P2.8 and P2.9, referred to in C as P2_0 and P2_1. Similarly, the corresponding direction bits are referred to as DP2_0 and DP2_1. Note that upon reset, the microcontroller ports are configured as inputs. We must set the direction control bit DP2_1 of the output bit P2.1 to configure it as an input. Once the port directions are configured, the program enters a loop. The loop counter nCount is incremented each time the push button PB0 is pressed and released. The output bit is toggled each time nCount is incremented, that is, has the state of the output is 0 if nCount is even and 1 id nCount is odd. The LED connected to the output is lit when the output bit is 0.

Note that the program spends quite a bit of time polling the input bit. Once pressed, the program must wait until PB0 is released, and similarly, once released, until PB0 is pressed again. A more efficient approach is to use the input as an interrupt source and write an interrupt service routine to toggle the output. This approach is discussed in the next section.

Single step through the program and observe its operation.

12.3.2 Polled Analog-to-Digital Conversion

This experiment uses the function AdcPoll from the module ADCPOLLUTILITIES.C under the archive project UTILITIES. ADCP8 is similar to the exercise given in section 11.3.1 above.

Hardware connections
Connect a few of the analog inputs of the microcontroller to the potentiometers POT0-POT8.

Software
Compile and run project ADCP8.

12.3.3 Simple LCD Operation

Liquid Crystal Displays (LCDs) are useful and powerful output devices since they allow plain text messages. Interacting with an LCD requires initializing the LCD and sending it various commands and data bytes. In this sense, an LCD is still an input/output device, one with a eight- or four-bit data interface and a 3-bit control interface. The commands are sent to the LCD by setting the data bits, and the appropriate control bits: R/W to write, and RS to command. Finally, the enable bit (E) is pulsed (momentarily made high. When using a 4-bit data interface, this operation is repeated twice per byte. Data bytes are sent to the LCD in a similar fashion, except that the RS line indicates data. Reading from the LCD is also similar, except that the control line R/W is set to the Read state.

Using a LCD is greatly simplified when you use existing archive utility functions. These functions are organized in module LCDUTILS.C in the archive project UTILITIES.

Hardware connections
We will use a 4-bit data bus. Connect the LCD data bus Data4-Data7 to the microcontroller ports P2.0 to P2.3 via a ribbon cable. Also connect the LCD signals RS, R/W, and E to the ports P2.4, P2.5, and P2.6, respectively.

Software
The project LCD.PRJ is provided for this experiment. The example software simply makes use of the archive functions LcdReset(), LcdClear(), LcdSendStr(), and LcdSendInt(). Try writing your own message on the LCD.

12.3.4 Combining Modules: ADC with LCD

This experiment is an exercise to combine a few archive utility functions to generate a powerful application. It is intended as an extension to the simple LCD operation example given above.

Hardware
1. Connect the LCD to port 2 bits as discussed above in section 5.1.3.
2. Connect POT0 to ADC channel 0

Software
Write a program to read the ADC channel 0 and report its value on the LCD.

You may start with the project LCD used in section 11.3.3. First save the LCD project under a different name. Then open the archive UTILITIES and drag the ADCPOLLUTILS into your project. Once the ADC utilities are available, use AdcPoll(0) to read the ADC value and LcdSendInt() to display it.

The user may refer to the project LCDADC which displays the ADC value until it exceeds 1000.

12.3.5 Stepper Motor Control

Stepper motors are convenient devices to generate motion. Stepper motors are especially suited to cases where the position control is required, such as in a disk drive head. From a control point of view, a stepper motor has 4 inputs which determine its step. The following generic pattern has 4 steps, which when completed corresponds to a given angle of rotation. In order to produce motion of a given angular distance, the steps are simply repeated the required amount.

Table 1. Full Step Sequence for Clockwise Rotation.

Step	Q1 P3.4, Red	Q2 P3.5, White	Q3 P3.6, Orange	Q4 P3.7, Blue	Equivalent Nibble Value
1	ON	OFF	ON	OFF	1010 = 10
2	ON	OFF	OFF	ON	1001 = 9
3	OFF	ON	OFF	ON	0101 = 5
4	OFF	ON	ON	OFF	0110 = 6
1 (repeat)	ON	OFF	ON	OFF	1010 = 10

In the table above, ON refers to an energized coil and OFF to an idle one. The RIO board has two stepper motor drivers (ST ULN2066B) which switch the coil currents based on TTL inputs. There is a separate TTL input for each coil, and an open collector-type output for each coil.

The table given above produces a full step of the motor each time the pattern is updated. It is possible to achieve half steps by inserting intermediate patterns into full steps. These intermediate states energize only one coil; the coil that is energized in the following and proceeding full steps.

Table 2. Half Step Sequence for Clockwise Rotation.

Step	Q1 P3.4, Red	Q2 P3.5, White	Q3 P3.6, Orange	Q4 P3.7, Blue	Equivalent Nibble Value
1	ON	OFF	ON	OFF	1010 = 10
2	ON	OFF	OFF	OFF	1000 = 8
3	ON	OFF	OFF	ON	1001 = 9
4	OFF	OFF	OFF	ON	0001 = 1
5	OFF	ON	OFF	ON	0101 = 5
6	OFF	ON	OFF	OFF	0100 = 4
7	OFF	ON	ON	OFF	0110 = 6
8	OFF	OFF	ON	OFF	0010 = 2
1	ON	OFF	ON	OFF	1010 = 10

Hardware connections

We will use the four bits of Port 3 (P3-4 to P3.7) to control the stepper motor. Connect these ports to the input of the stepper motor driver marked “Stepper A.” Connect the common of the stepper motor driver to VCC. Connect the outputs of the stepper motor driver to the stepper motor following the color codes in the tables above. Also connect the common supply lines of the stepper motor (two brown wires) to VCC.

Software

The project STEPPER.PRJ is provided for this experiment. The program is exceedingly straightforward. A simple loop is executed for a fixed number of steps. A new pattern is sent to the stepper motor at each software loop step. A simple delay function StepDelay() is called between the steps. The duration of this delay function determines the stepper motor speed.

Note that this project does not contain any of the previously used archive utility functions. You may add a feature that displays the current step on the LCD by simply dragging and dropping the LCDUTILITIES.C module from the UTILITIES archive. Then report the current step by simply using the LcdSendInt() function. The users may refer to the project STEPLCD for an implementation.

You will note that using the simple delay function StepDelay() to determine motor speed is not the most accurate approach if the foreground program needs to tackle other tasks such as displaying the step on the LCD. The motor speed is reduced when the LCD steels the attention of the microcontroller. There is a better approach. A timer may be set up to generate periodic interrupts. The interrupt service routine may then update the step. This way, even if the foreground program is busy with other tasks, the CPU is interrupted by the higher priority interrupt, and thus, the motor speed does not fluctuate. This approach is presented below, after interrupt routines are discussed.

12.4 Interrupts and Real-Time Programming Architectures

Interrupt processing in C is a fundamental task of Rc66. Interrupts provide the necessary hardware elements to satisfy the requirements of real-time applications.

Servicing Simple Interrupt Signals
Interrupt-Driven Analog-to-Digital Conversion
Accurate Timing Routines Using Timer Interrupts
Accurate Stepper Motor Speed Control Using Timer Interrupts

12.4.1 Servicing Simple Interrupt Signals

Revisit the experiment given in section 11.3.1. It was mentioned that the CPU spends much valuable cycles polling the push button. A more efficient approach is now given.

<u>Hardware connections</u>
The hardware set up is the same as in the experiment given in section 11.3.1. Port P2.8 is used as an input and P2.9 as an output. Connect push button PB0 of the RIO board to P2.8 of the microcontroller. Also connect P2.9 to LED0 of the RIO board.

<u>Software</u>
Load the project PIOINT. PIOINT consists of three modules, a main code module that is given below, and two utility modules S0.C and INTUTILITIES.C from the archive project UTILITIES.

```
#define LIMIT 10

predef bit P2_8,   // Port bit P2.8
           P2_9,   // Port bit P2.9
           DP2_8,  // Port direction bit DP2.8
           DP2_9;  // Port direction bit DP2.9

predef unsigned int CCM2, CC8IC;

int nCount=0;      // global variables are visible from Isr

main(void){
int nOldCount=0;

// --- set up port bits and directions ---
 DP2_8=0;          // P2.8 is an input
 DP2_9=1;          // P2.9 is an output
// --- send title string ---
 SendStr("\n\nInterrupt-driven I/O example\n\n");

// --- set up interrupt ---
 SetIntVec(0x60, Isr);
 CCM2&=0xFFF0;
 CCM2|=2;        // CCMOD8=2 (interrupt on neg edge)
 CC8IC=0x4C;     // enable CAPCOM8 interrupts - low priority
 IEN=1;          // turn master interrupt enable on

// --- this loop repeats 'LIMIT' times as defined above
 do
  {
   P2_9=nCount%2; // toggle LED at each button press

   // --- display current count ---
   SendStr("\nCount : ");
   SendInt(nCount);

   // --- wait for the next interrupt ---
   while(nCount==nOldCount);
   nOldCount=nCount;         // increment old count...
  } while(nCount<LIMIT);

// --- wrap up ---
 DP2_9=0;        // P2.9 is an input
 IEN=0;          // turn master interrupt enable off
 CC8IC=0;        // disable CAPCOM8 interrupts

 SendStr("\n\n");
}
/* -------------------- */
void interrupt Isr(void){
 nCount++;  // increment count
}
```

An interrupt-driven approach requires the following steps:

1. Writing the interrupt service routine
2. Setting the interrupt vector and interrupt priority
3. Enabling the interrupt
4. Optionally, initiating the process
5. Cleaning up after the process is completed.

Since interrupt service routines are called upon hardware signals, they cannot have arguments or return a value. Consequently, any information exchange between the foreground code and interrupt service routines must be accomplished by global variables that are visible to all parts of the program. In this example, the interrupt service routine uses the global variable nCount to pass information to the foreground program.

The keyword interrupt instructs the compiler that the function must be written as an interrupt service routine. This essentially means pushing and popping all relevant registers and returning with an RETI instruction. Rc66 also uses a different expression stack set up mechanism for interrupt service routines.

We use CAPCOM8 to generate interrupts on the falling edge of the input. Thus the lowest nibble of CCM2 is set to 2. Each time the push button PB0 is pressed, and interrupt will be generated.

The 166 family of microcontrollers have dedicated interrupt vectors for the various hardware interrupts. A jump instruction to the interrupt service routine must be placed at these predefined vectors. The function SetIntVec() sets the interrupt vector. SetIntVec() takes two arguments: the vector and the address of the interrupt service routine. Note that in this example, the P2.8 (CAPCOM8) interrupt has vector address 60h. Similarly, the interrupt must be enabled. The CAPCOM interrupt control register CC8IC is initialized to 4Ch, which sets its enable flag and assigns it an interrupt priority of 3 with group level 0. Although we enabled the CAPCOM0 interrupt, the CPU will not process interrupts unless the master interrupt enable flag IEN located in the PSW register is set. Simply set IEN=1 to allow interrupts.

Upon termination, CAPCOM8 interrupts are disabled by clearing the CC8IE flag and giving it priority 0. This is accomplished by setting CC8IC to 0. Similarly, the master interrupt enable flag IEN is cleared.

It is recommended that you place a break point in the interrupt service routine and run your program. Press the push button and observe that the program stops in the interrupt service routine and increments the count. You may observe the global variable values while the program is in the foreground loop or in the interrupt service routine.

The support function SetIntVec() is placed in the archive UTILITIES in module INTUTIL.C. You may use this function in your applications, or remove the debug suppression directive and single step through SetIntVec() to inspect its operation.

12.4.2 Accurate Timing Routines Using Timer Interrupts

Accurate time keeping is the foundation of real-time programming. The 166 family of microcontrollers have a wealth of timers which makes implementing a time base a simple task.

Hardware connections

No external hardware is needed for this experiment.

Software

Open the project TIMER. The project consists of three modules, a main code module and the now familiar support utility modules S0.C and INTUTILITIES.C from the archive project UTILITIES.

The program is given below. The reload value of Timer0 forces overflows every millisecond, at which time an interrupt is generated. The interrupt service routine for Timer0 updates the ticks (milliseconds) and the time in seconds. The two words, MSEC and SECOND store the milliseconds and the seconds respectively.

```
// --- constants ---
// 0xF63C=10000h-2500
// reload value for overflows every 1 millisec
// with CPU clock=20MHz (50 nanosecond clock)
// and the prescalar set to 8.  That is,
// 8*50*2500=1000000 nanoseconds or 1 millisecond
#define MSEC_COUNT  0xF63C


// --- global variables ---
unsigned int MSEC;             // milliseconds, low word
unsigned int SECONDS;          // seconds

predef unsigned int T0, T0REL, T01CON, T0IC, S0RIC;
predef unsigned bit T0R, S0RIR;
```

```
unsigned int nSeconds=0;

main(void){
 MSEC=0;
 SECONDS=0;

 SetIntVec(0x80, T0Isr); // assign interrupt service routine

 S0RIC=0;
 T0init(MSEC_COUNT);     // initialize Timer0
 IEN=1;                  // enable interrupts


// --- start the main loop - press CTL-C to quit ---
 do
  {
   if(SECONDS>nSeconds)
    {
     nSeconds=SECONDS;
     SendStr("\nTime: ");
     SendInt(SECONDS);
     }
   } while(SECONDS<10);
 T0IC=0;
}

// ---------------------------------------------------------
// subroutine T0init initializes Timer0
// input    : timer 0 reload value
// output   : none
// ---------------------------------------------------------
void T0init(int nReload){

T0REL=nReload;         // reload value determines the period
T0=nReload;            // start timer with reload value

// --- set up the timer control register ---
 T01CON=0;             // prescalar=8, Timer0 off
// T01CON=7;           // prescalar= 1024, Timer0 off
 T0IC=0x45;            // enable T0 interrupts - priority
                       // level=1, group=1

// --- initialize variables ---
 MSEC=0;                // initialize ticks low word
 SECONDS=0;             // initialize seconds

// --- start timer ------------
 T0R=1;                 // start timer
}

// ---------------------------------------------------------
// subroutine T0Isr Timer0 Interrupt Service Routine
//
// called upon Timer0 overflows
// ---------------------------------------------------------
void interrupt T0Isr(void){
int i;
 MSEC++;
 if(MSEC>=1000)
  {
   MSEC=0;
   SECONDS++;
   }
}

// ---------------------------------------------------------
// end of program...
```

The foreground loop polls SECONDS and displays its value every time it changes. Note that the timing is extremely accurate (as accurate as the 40MHz crystal oscillator).

Although the structure of this program is no different than the other examples with interrupt service routines, it requires a few extra steps to set up the timer and the related global variables. These steps are placed in a separate function, T0Init(), which takes the reload value as an argument and starts Timer T0. Also note that the interrupt service routine is a lot longer than the one or two line ones given in previous examples. Although one should endeavor to write the interrupt service routines as short as possible, the one used in this example by no means pushes the capabilities of the microcontroller.

12.4.3 Accurate Stepper Motor Speed Control Using Timer Interrupts

It was mentioned in section 11.3.5 that the foreground task of displaying the current step places demands on the CPU which results in a fluctuating stepper motor speed. With an interrupt-driven time base, we may now tackle the problem of generating smooth stepper motor speeds even in the event of foreground tasks.

Hardware connections
The hardware set up is the same as given in section 11.3.5. We use the four bits of Port 3 (P3-4 to P3.7) to control the stepper motor. Connect these ports to the input of the stepper motor driver marked “Stepper A.” Connect the common of the stepper motor driver to VCC. Connect the outputs of the stepper motor driver to the stepper motor following the color codes in the tables above. Also connect the common supply lines of the stepper motor (two brown wires) to VCC.

Software
Open the project TIMEDSTEPPER.PRJ. This project builds upon the previous example TIMER.PRJ. Stepper motor functions HalfStep() and FullStep() are included along with the two functions StepperReset() and StepperOff() to set up the corresponding port bits.

This project differs from the previous timer project in its interrupt service routine. The interrupt service routine is given below. Rather than keeping time, the focus here is to increment the motor step at regular intervals, determined by the constant STEPPERIOD. In the example, STEPPERIOD is set to 10 milliseconds.

```
void interrupt T0Isr(void){
int i;
 MSEC++;
 if(MSEC>=STEPPERIOD)
  {
   MSEC=0;
   FullStep(ngStep++);
   }
}
```

The foreground loop simply waits until all 1000 steps are completed. This should take exactly 10 seconds. Run the program and then add the LCD functions to display the current step. Note that the LCD operation should not cause motor speed fluctuations. The user may review the project TIMEDLCDSTEPPER for an implementation.

12.4.4 Interrupt-Driven Analog-to-Digital Conversion

Interrupt service routines are the key to real-time software architectures. Real-time refers to prompt actions taken upon receiving external signals. Of course, a real-time system should not be overloaded to the point that it becomes too busy to react to such external signals.

Hardware connections
Connect POT0 to the microcontroller ADC channel 0.

Software
The project ADCISR is given for this experiment. Almost all interrupt-driven software applications will follow a similar structure. The main program is presented below illustrates this structure.

```
// --- global variables are visible from ISRs ---
int nADC, bFlag;

main(void){
int nCount;

 SetIntVec(0xA0, AdcIsr); // assign interrupt service routine
 ADCIC=0x4C;              // enable ADC interrupts - low priority
 IEN=1;                   // enable interrupts

 bFlag=0;
 nCount=0;
 ADCON=0x80;              // start ADC channel 0

 do
  {
   while(!bFlag);
   bFlag=0;
   SendStr("\n ADC Channel 0: ");
   SendInt(nADC);
   nCount++;
   ADCON=0x80;           // start ADC channel 0
   } while(nCount<10);

 IEN=0;                   // disable interrupts
 ADCIC=0;                 // disable ADC interrupts
}

// ------------------------------------------------------------
// Analog-to-Digital Conversion Interrupt Service Routine
// ------------------------------------------------------------
void interrupt AdcIsr(void){
 nADC=ADDAT;             // copy ADC value
 bFlag=1;                // signal ADC completion
}
// ------------------------------------------------------------
```

The interrupt service routine uses the global variable nADC to return the conversion value. The synchronization of the foreground program to the interrupt service routine is also accomplished by global variables. In this example the Boolean flag bFlag is used to signal that the ADC conversion is completed. The foreground loop pools the flag and uses the ADC value only when the flag is set. The foreground loop clears the flag before it requests another conversion by writing 80h to ADCON.

The ADC completion interrupt has vector address A0h. Similarly, the interrupt must be enabled. The ADC interrupt control register ADCIC is initialized to 4Ch, which sets its enable flag and assigns it an interrupt priority of 3 with group level 0. An ADC conversion is initiated by writing 80h to ADCON. This starts converting channel 0 in a fixed single channel conversion mode. Each time a conversion is needed, the value 80h is written to ADCON.

Upon termination, the ADC interrupts are disabled by clearing the ADCIE flag and giving it priority 0. This is accomplished by setting ADCIC to 0. Similarly, the master interrupt enable flag IEN is cleared.

It is recommended that you single step through the program and observe that the interrupt service routine is invoked following ADC conversions. You may observe the global variable values while the program is in the foreground loop or in the interrupt service routine.

APPENDICES

APPENDIX A: READS166 3.1 UPDATES

Reads166 V3.10 introduces the following updated features over version 3.0x.

1. A completely rewritten Integrated Development Environment (IDE) and user interface.
2. A more streamlined C compiler with (4-byte) floating point and long integer support. Some support for C structures, unions, and enumerated types.
3. ST10F262/ST10F272 Multiply-Accumulate (MAC) instructions are added to the assembler.

Bug Reports:
Please report the bugs in this version to Rigel Corporation. The preferred method is by e-mail to "tech@rigelcorp.com"

A.1 Project Formats

Reads V3.10 project database is different from the V3.00 database. The IDE will open Reads166 V3.00 projects and convert to the new format. The old projects are saved in a subdirectory in case you would like to revert back.

A.2 Hardware Configuration

ReadsV3.00 attached hardware configuration information to individual projects. Although this allows the same code to be implemented in two different projects for, say, the 166 and 167 processors, in some cases, it also lead to ambiguities. Reads166 V3.10 views the hardware configuration as global information, identical across all projects. Once the hardware configuration is set from the menu, it is applied to all projects as well as to the compilation of projectless source code.

A.3 User Configurable Syntax Highlighting

You may add your keywords to the list to be highlighted. Activate the editor and look under the Edit / Colors submenu.

A.4 Matching Parentheses

Place the cursor next to a parentheses in the editor. Press the right mouse button and select the menu item "Find Matching {" or use the hotkey ALT-P.

A.5 Search Across Files

This "grep-like" utility scans the specified directories for the occurrences of given keywords. Look under the Tools menu for Keyword Search.

A.6 Drag-and-Drop

Besides transferring modules between projects as before, the drag-and-drop feature now supports source files to be dragged from the other utilities, such as the Windows Explorer or the Windows Find dialogs into the project.

A.7 Project Notes

You may include files into the project but exclude them from build. This is a convenient way to attach text files to projects and keep notes. Check the "Exclude from build" box under the module options.

A.8 Variable and SFR Watches

The new watches have different tabs for different lists of variables. One of the tabs lets you create a set of selected variables. Use the right mouse button for local options, such as decimal or HEX display.

A.9 Floating-Point Numbers

Look at the projects Sin01, and Factorial_Float for examples.

A.10 Long Integers

Look at the project Factorial_LongInt for an example.

A.11 MAC Instructions

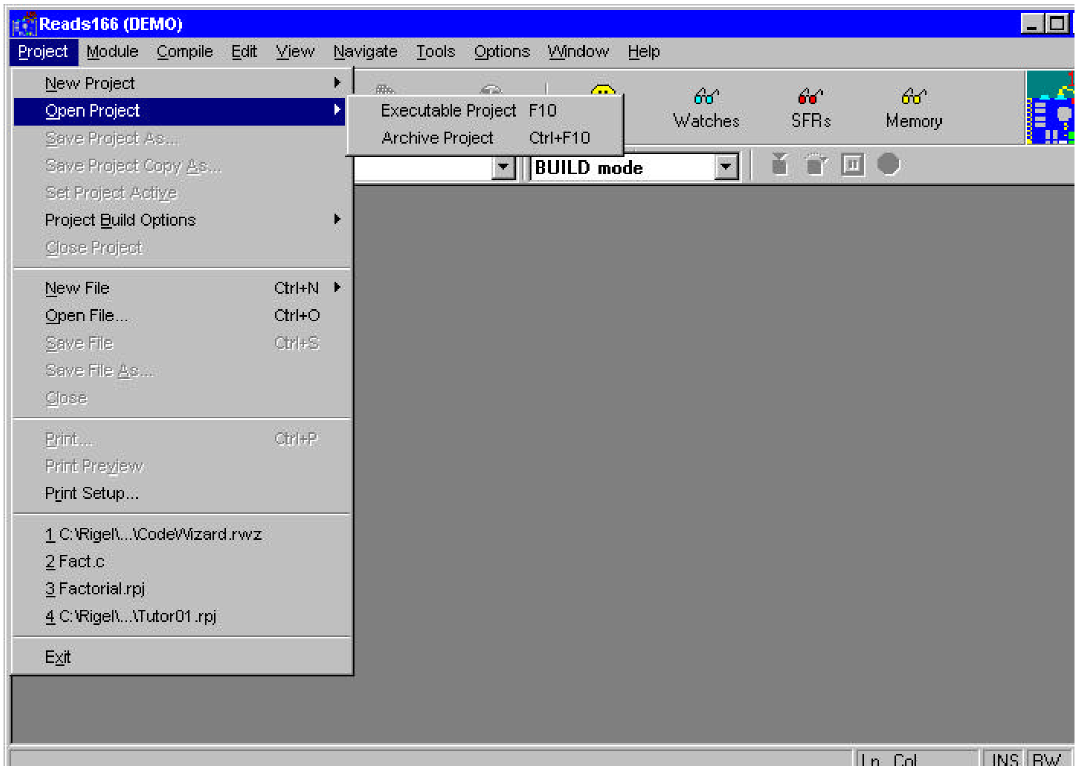
Enable the MAC262.272 instructions using the checkbox in the "Assembly Options" dialog. See the project MAC262 for examples.

APPENDIX B: READS166 MAIN MENU COMMANDS

The Main Menu contains the higher-level options such as projects, modules, or tools.
The major tasks are delegated to the READS166 "**Tools**" which includes editors, host-to-board communications subsystems and code generators. Tools are distinguished by their own environments including sub-menus and accelerator keys. Tools may be minimized when not used or simply closed until needed again.

B.1 Project

Projects are collections of source code modules that are compiled as a whole. Use the project menu to create **"New"** projects, **"Open"** existing projects, **"Close"** projects, select project **"Options"**, and save projects. The "**Exit**" command is also under the menu "**Project**" option.



The use of projects is optional in READS166. It is meant to simplify the bookkeeping of the various components of larger code. For short programs, it is often more practical to simply write the code in the text editor and compile it without first creating a project. (The demo version has limitations on the types and sizes of projects.)

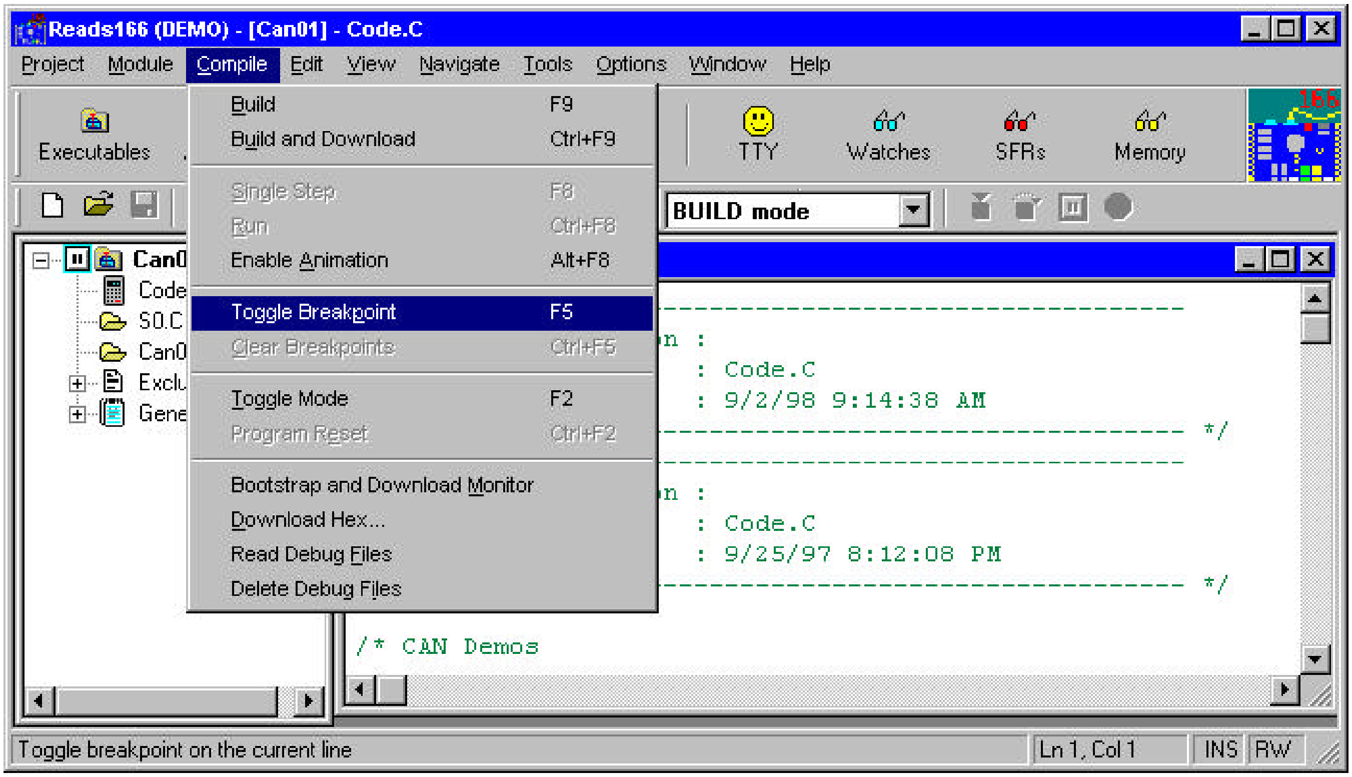
The Project Window is the space just under the Main Menu. If a project is currently open, a list of modules of the project is displayed. You may use the scroll bars to view the module list.

B.2 Module

A module is a single file that belongs to a project. Typically modules are either assembly language subroutines or C language functions. You may copy modules from one project to another, or share modules in different projects. For example, you may copy a previously developed module from an archive project to an executable project by simply dragging its icon from one project window to the other. A new feature of READS166 Version 3 is that you may combine C and assembly modules in one project to decrease code size and execution time.
You may set "**Module Properties**", "**Add Module**", "**Edit Module**", "**Save**", "**Save All**", or "**Delete**" modules of the current project using the commands under the "**Module**" option. The Add Module command allows you write and include new modules to a project, or lets you select assembly or C programs that are currently in an archive project to be included in the current project. There is also a selection called "**Code Wizard**" in the module options. The Code Wizard opens to show a C Wizard or an ASM Wizard. Both of these wizards generate code automatically for your project. READS166 does not require the use of modules.

B.3 Compile

The "**Compile**" menu commands allow you to "**Toggle Mode"** between the Build Mode and Run / Debug Mode. In the Build Mode projects or modules are compiled. In the Run / Debug Mode the built projects are downloaded to the board. "**Build"** compiles the current project, or compiles a single module depending on which is highlighted in the Project Window. If the editor contains a file, this current file is compiled. "**Build and Download"** compiles the highlighted project and downloads it to the target board.



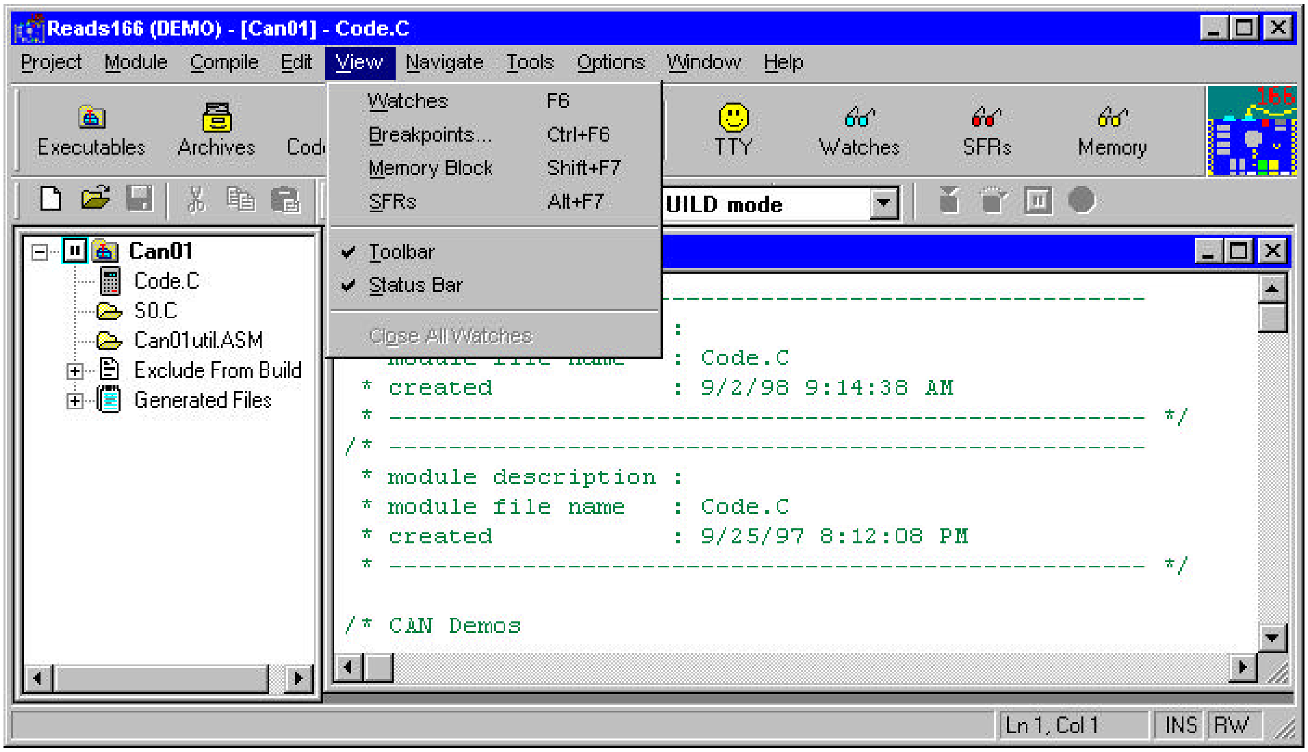
```
-----------------------------------
n :
  : Code.C
  : 9/2/98 9:14:38 AM
----------------------------------- */
-----------------------------------
n :
  : Code.C
  : 9/25/97 8:12:08 PM
----------------------------------- */

/* CAN Demos
```

Toggle breakpoint on the current line

"Clear Breakpoints" removes all breakpoints from your selected program. "**Toggle Breakpoints"** allows you to turn on or off selected breakpoints. The "**Single Step"** command allows you to step through the program, statement by statement, after it has been downloaded to the board. The "**Enable Animation"** command animates the singlestep feature. Once the singlestep command is issued the enable animation automatically continues to singlestep from statement to statement without pushing buttons. **"Bootstrap"**, **"Bootstrap and Download Monitor"**, "**Download Hex"**, **"Read Debug Files"** and **"Delete Debug Files"** are basic features that implement the stated commands. The "**Run"** command runs the downloaded program on the target board. The Bootstrap selection bootstraps the board preparing it for downloading files.

B.4 View

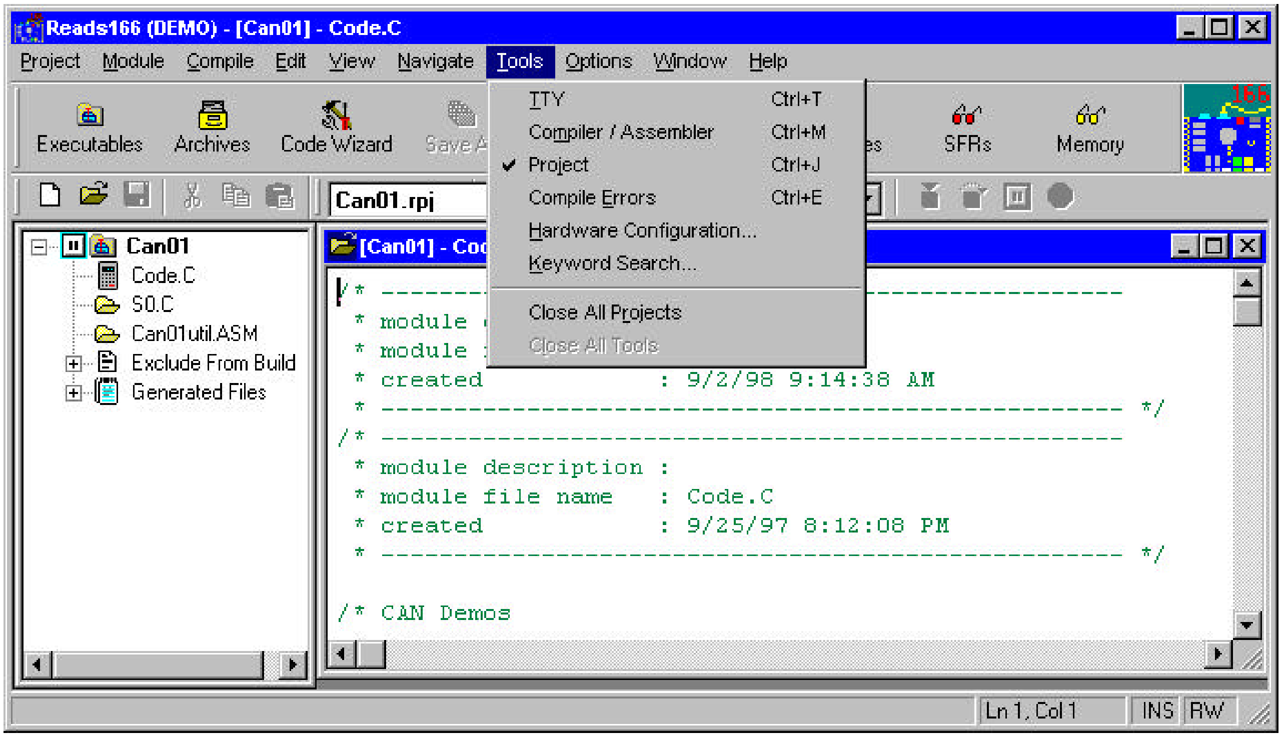
The **"View"** menu commands allow the user to watch the variables and breakpoints as the program is debugged. The **"Watches"** menu command also allows you to select the variables that you wish to watch.



```
 -----------------------------------
 :
 module file name   : Code.C
 * created            : 9/2/98 9:14:38 AM
 * ---------------------------------------------------- */
/* ----------------------------------------------------
 * module description :
 * module file name   : Code.C
 * created            : 9/25/97 8:12:08 PM
 * ---------------------------------------------------- */

/* CAN Demos
```

B.5 Tools

Tools are the powerful subsystems that let you carry out complicated tasks. Tools usually have their own menus and hot-key combinations. Currently the following tools are available **"TTY", "Compiler/ Assembler", "Project", "Compile Errors"**, "**Hardware Configuration", "Keyword Search", "Close All Projects",** and **"Close All Tools".**



```
/* -------
 * module
 * module
 * created            : 9/2/98 9:14:38 AM
 * ---------------------------------------------------- */
/* ----------------------------------------------------
 * module description :
 * module file name   : Code.C
 * created            : 9/25/97 8:12:08 PM
 * ---------------------------------------------------- */

/* CAN Demos
```

B.5.1 TTY

The TTY Window encapsulates all host-to-board communications. It has its own menu to set the communications parameters, to bootstrap the board, and to download compiled programs into the RAM of the board.

B.5.2 Compiler/ Assembler

The Compiler / Assembler command opens the C compiler window which allows you to select between the C compiler and the assembler. It also allows you to set the C and assembly language parameters from the Tools | Compiler/ Assembler | Options menu.

B.5.3 Project

The Project selection toggles the Project window open or closed.

B.5.4 Compile Errors

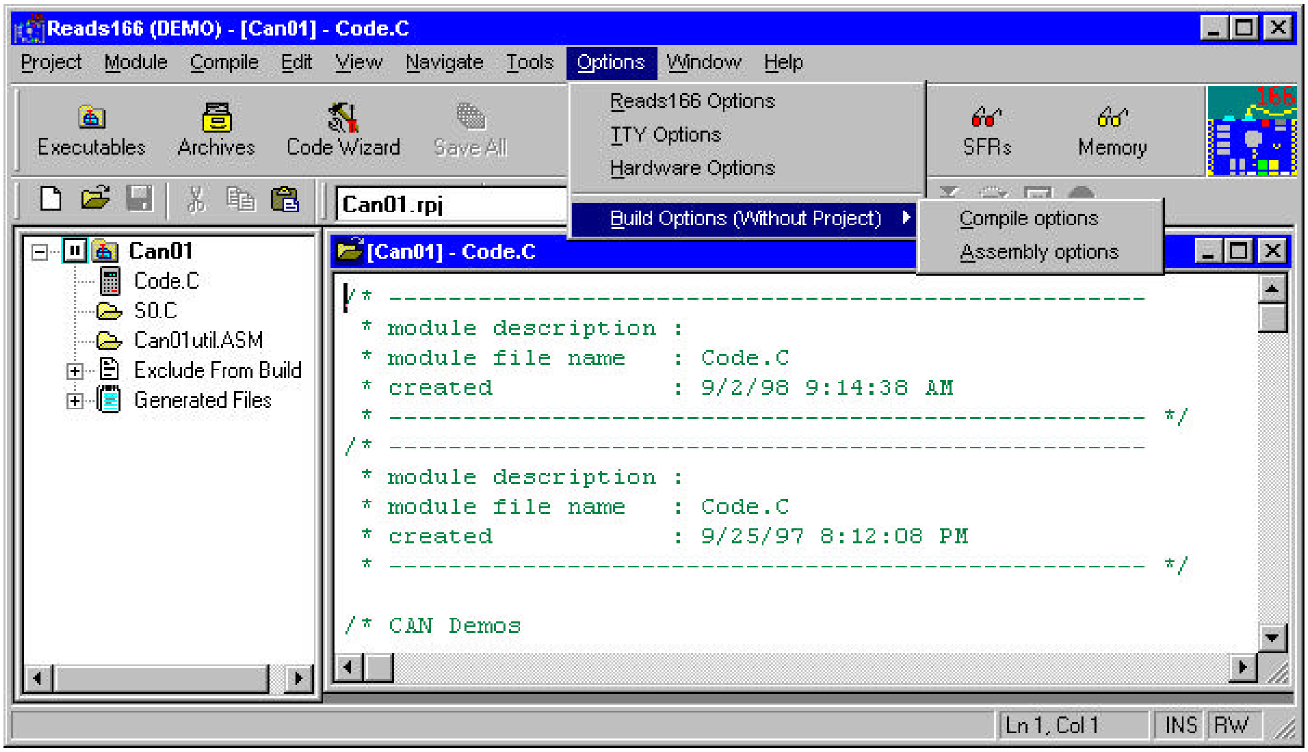
Selecting the Compile Errors command toggles the READS Error Dialog window off and on.

B.5.5 Hardware Configuration

The Hardware Configuration command opens a window where you may set the configuration of the software according to the hardware platform you are working on. By using the FIND button you can easily select the correct hardware which will then set up the rest of the parameters listed in the window.

B.6 Options

There are four commands in the "**Options"** menu. The "**READS166 Option"**, allows you to select time-outs and whether to terminate the compiler after use. The "**TTY Options"** allow you to select the COMM port, Baud rate, and TTY Time-outs. "**Hardware Options"** allow you to select the hardware configuration**.** The fourth option is for use when you are not using projects for code development. The "**Build Options"** opens up a submenu to select compiler or assembly options available when not using projects.



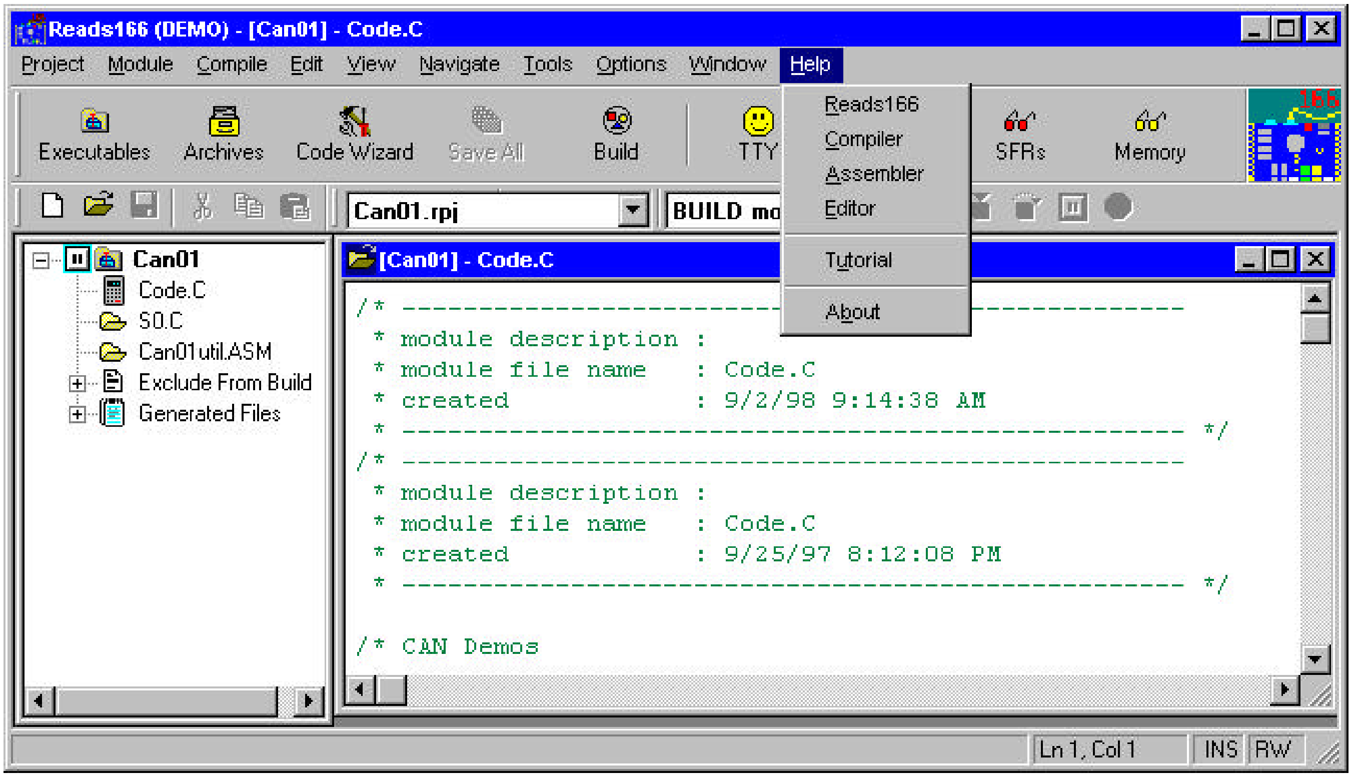
```
/* ---------------------------------------------------
 * module description :
 * module file name    : Code.C
 * created             : 9/2/98 9:14:38 AM
 * --------------------------------------------------- */
/* ---------------------------------------------------
 * module description :
 * module file name    : Code.C
 * created             : 9/25/97 8:12:08 PM
 * --------------------------------------------------- */

/* CAN Demos
```

B.7 Window

The Window command allows you to select how the READS166 windows will appear.

B.8 Help

This command invokes the READS166 Help system.



```
/* ------------------------          ---------------
 * module description :
 * module file name    : Code.C
 * created             : 9/2/98 9:14:38 AM
 * --------------------------------------------------- */
/* ---------------------------------------------------
 * module description :
 * module file name    : Code.C
 * created             : 9/25/97 8:12:08 PM
 * --------------------------------------------------- */

/* CAN Demos
```

APPENDIX C: READS EDITOR

C.1 READS166 Editor Overview

The Reads Editor implements a standard text editor with menus and controls familiar to most Windows applications. The menu items and the actions taken by them are described below.

The Reads Editor is not a stand-alone application. It is a part of the Integrated Development Environment (IDE). The editor may be turned on and off from the Tools menu or by the hot key combination Ctrl+E. In some cases, the IDE automatically opens the editor, for example, to display a break point during a debug session.

The Reads Editor is more than just a text editor. Its behavior depends on the current state of the IDE as listed below.

IDE State	Reads Editor Tasks
Writing code	Text editor, keyword help
Compiling/assembling	Show errors
Debugging	Show current step

In addition, the Reads Editor supports a local popup menu, activated by clicking the mouse right button within the editor. All editor tasks and links, such as building the current project or clearing all breakpoints during a debug session may be initiated by selecting the corresponding popup menu item.

C.2 File Menu

This menu groups the operations which deal with storing, retrieving or printing files. It also keeps a history list of the last few files opened, so that they may quickly be reopened.

New	Ctrl+N	Opens a new file.
Open	Ctrl+)	Opens an existing file.
Close		Closes the current file.
Save	Ctrl+S	Saves the current file to disk.
Save As		Saves the current file under a different file name.
Print	Ctrl+P	Prints the current file.
Print Preview		Displays the page as it will be printed.
Printer Setup		Selects printer options.
Exit		Terminates the editor.

C.2.1 Edit Menu

The editor uses the clipboard to facilitate copying segments of text from one place to another. Such text is first tagged by highlighting it. You may highlight text by dragging the mouse while keeping the left button down, or from the keyboard, by using the arrow keys while holding the shift key down.

Cut	Ctrl+X	Removes the highlighted text from the file and places it into the clipboard.
Copy	Ctrl+C	Copies the highlighted text into the clipboard without removing it from the file.
Paste	Ctrl+V	Places the contents of the clipboard into the file at the current carret position.
Undo	Ctrl+Z	Restores the document to its state immediately before the last edit command.
Select All	Ctrl+A	Selects the contents of the entire file.

C.2.2 View

You may change the appearance of the editor by the following commands.

Toolbar	Displays a set of icons at the top of the editor window. These are shortcuts to the more often used menu commands. Currently the toolbar includes the following shortcuts: File New, File Open, File Save, Cut, Copy, Paste, Print, and Help.
Status Bar	Displays a bar at the bottom of the editor window that shows the current line and column. It also displays the status of the following keys: Overwrite, CapsLock, and NumLock.
Status Ribbon	The status ribbon is an alternative form of the status bar.
8 pt	Selects the smallest fonts.
10 pt	Selects the medium size fonts.
12 pt	Selects the largest size fonts.

C.2.3 Window

Cascade	Arranges all editor windows in a cascade fashion.
Tile	Tiles all editor windows. This is especially usefult to view two files simultaneously.
Arrange Icons	You may arrange the minimized edit windows neatly by this command.

C.2.4 Navigate

Navigation commands move the caret within the text file. Although many of the navigation commands are available from the menu, in most cases, the hot key combinations are more convenient. For example, use Ctrl+Home to bring the caret to the beginning of the file.

In addition to the menu commands listed below, several keys (see Miscellaneous Edit and Navigation Keys) move the caret, sometimes while performing another operation. For example, the backspace key moves the caret backwards one character while erasing that character.

Search		Moves the caret to the next occurrence of the given string.
Search Forward		Repeats the last search in the forward direction.
Search Backward		Repeats the last search in the backward direction.
Replace		Replaces the next occurrence of the search string with another string.
Beginning of Line	Home	Moves the caret to the beginning of the line.
End of Line	End	Moves the caret to the end of the line.
Beginning of File	Ctrl+Home	Moves the caret to the beginning of the file.
End of File	Ctrl+End	Moves the caret to the end of the file.
Next Word	Ctrl+Right	Moves the caret to the next word.
Previous Word	Ctrl+Left	Moves the caret to the previous word.
Jump		Jumps to a given line number.

C.3 Miscellaneous Edit and Navigation Keys

Up Arrow	Moves caret to the previous line.
Down Arrow	Moves caret to the next line.
Right Arrow	Moves caret to the next character.
Left Arrow	Moves caret to the previous character.
PgUp	Moves caret to the previous page.
PgDn	Moves caret to the next page.
Del	Deletes the next character.
Backspace	Deletes the previous character.

C.4 Highlighting Text

Highlighting text is the most fundamental operation in using the edit commands. There are various shortcuts to highlighting text, also referred to as selecting text.

C.4.1 Highlighting the Current Word

Position the mouse cursor on the word to be highlighted and double click any mouse button.

C.4.2 Highlighting a Block of Characters

Using the mouse

Position the mouse cursor on the first character of the block and depress the left button. While the left button is depressed, drag the mouse to the last character of the block and release the mouse.

Using the Keyboard

Position the caret on the first character of the block and depress the shift key. Now, use the arrow keys to position the caret on the last character of the block and release the shift key. Normally, you can use any position key in combination with the shift key to advance or shrink the highlighted area.

Normally a function that utilizes a character block also erases the highlighting. To explicitly erase the highlighting click the mouse one more time or press any of the positioning keys.

C.4.3 Highlighting a Block of Lines

Using the mouse

Position the mouse cursor at any position on the first line of the block and depress the right button. While the right button is depressed, drag the mouse to the last line of the block and release the mouse.

Using the Keyboard

Position the caret at any position on the first line of the block and hit the F8 function key. Use the Up and Down arrow keys to position the caret on the last line and hit F8 again.

Normally a function that utilizes a line block also erases the highlighting. To explicitly erase the highlighting, click the mouse one more time or press one of the arrow keys.