R-515JC / Reads51
USER'S GUIDE

Version 2.02
December 2002

Rigel Corporation
P.O. Box 90040, Gainesville, Florida
(352) 373-4629, FAX (352) 373-1786
www.rigelcorp.com, tech@rigelcorp.com

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.

HARDWARE WARRANTY

Limited Warranty. Rigel Corporation warrants, for a period of sixty (60) days from your receipt, that READS software, RROS, hardware assembled boards and hardware unassembled components 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. Although Rigel Corporation warranty covers 60 days, Rigel shall not be responsible for malfunctions due to customer errors, this includes but is not limited to, errors in connecting the board to power or external circuitry. This warranty does not apply to products which have been subject to misuse (including static discharge), neglect, accident or modification, or which have been soldered or altered during assembly and are not capable of being tested.

DO NOT USE PRODUCTS SOLD BY RIGEL CORPORATION AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS!
Products sold by Rigel Corporation are not authorized for use as critical components in life support devices or systems. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

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 RIGEL CORPORATION PRIOR TO USE.

Return Policy. This policy applies only when product purchased directly from Rigel Corporation. 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. This does not apply to books. Books are non-returnable.
Please call (352) 373-4629 to receive an RMA (Returned Merchandise Authorization) number prior to returning product. You will be responsible for shipping costs.
All returns must be made within 30 days of date of invoice and be accompanied by the original invoice number and a brief explanation of the reason for the return.
Return merchandise in original packaging.
All returned products are subject to a $15 restocking charge. "Custom Items" are not returnable.

Repair Policy. If you encounter problems with your board or software after the 60 day warranty period, please call Rigel Corporation at (352) 373-4629 or email tech@rigelcorp.com for advice and instruction.

Rigel Corporation will test and attempt to repair any board. You will be responsible for shipping costs and repair fees. If you send a detailed report of the problems you encountered while operating the board, Rigel Corporation will inspect and test your board to determine what the problem is free of charge. Rigel Corporation will then contact you with an estimated repair bill. You will have the choice of having the board fixed, returned to you as is, or purchasing a new board at a reduced price. Rigel Corporation charges repair fees based on an hourly rate of $50.00. Any parts that need to be replaced will be charged as separate items. Although Rigel Corporation will test and repair any board, it shall not be responsible for malfunctions due to customer errors, this includes but is not limited to, errors in connecting the board to power or external circuitry.

Board Kit. If you are purchasing a board kit, you are assumed to have the skill and knowledge necessary to properly assemble same. Please inspect all components and review accompanying instructions. If instructions are unclear, please return the kit unassembled for a full refund or, if you prefer, Rigel Corporation will send you an assembled and tested board and bill you the price difference. You shall be responsible for shipping costs. The foregoing shall apply only where the kit is unassembled. In the event the kit is partially assembled, a refund will not be available, however, Rigel Corporation can, upon request, complete assembly for a fee based on an hourly rate of $50.00. Although Rigel Corporation will replace any defective parts, it shall not be responsible for malfunctions due to errors in assembly. If you encounter problems with assembly, please call Rigel Corporation at (352) 373-4629 for advice and instruction. In the event a problem cannot be resolved by telephone, Rigel Corporation will perform repair work, upon request, at the foregoing rate of $50.00 per hour.

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

Rigel Corporation's Software License Agreement

This Software License Agreement ("Agreement") covers all software products copyrighted to Rigel Corporation, including but not limited to: Reads51, rLib51, RbHost, RitaBrowser, FLASH, rChpSim, Reads166, and rFLI.

This Agreement is between an individual user or a single entity and Rigel Corporation. It applies to all Rigel Corporation software products. These Products ("Products") includes computer software and associated electronic media or documentation "online" or otherwise.

Our software, help files, examples, and related text files may be used without fee by students, faculty and staff of academic institutions and by individuals for non-commercial use. For distribution rights and all other users, including corporate use, please contact:
Rigel Corporation, PO Box 90040, Gainesville, FL 32607
or e-mail tech@rigelcorp.com

Terms and Conditions of the Agreement

1. These Products are protected by copyright laws, intellectual property laws, and international treaties. Rigel Corporation owns the title, copyright, and all other intellectual property rights in these Products. We grant you a personal, non-transferable, and non-exclusive license to use the Products. These Products are not transferred to you, given away to you or sold to you.

   Non-commercial use: These Products are licensed to you free of charge.

   Commercial use: You must contact Rigel Corporation to find out if a licensing fee applies before using these Products.

2. You may install and use an unlimited number of copies of these Products.

3. You may store copies of these Products on a storage device or a network for your own use.

4. You may not reproduce and distribute these Products to other parties by electronic means or over computer or communication networks. You may not transfer these Products to a third party. You may not rent, lease, or lend these Products.

5. You may not modify, disassemble, reverse engineer, or translate these Products.

6. These Products are provided by Rigel Corporation "as is" with all faults.

7. In no event shall Rigel Corporation be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use of or inability to use the Product, even if Rigel Corporation has been advised of the possibility of such damages. Because some states do not allow the exclusion or limitations of consequential or incidental damages, the above limitations may not apply to you.

8. Rigel Corporation makes no claims as to the applicability or suitability of these Products to your particular use, application, or implementation.

9. Rigel Corporation reserves all rights not expressly granted to you in this Agreement.

10. If you do not abide by or violate the terms and conditions of this Agreement, without prejudice to any other rights, Rigel Corporation may cancel this Agreement. If Rigel Corporation cancels this Agreement; you must remove and destroy all copies of these Products.

11. If you acquired this Product in the United States of America, this Agreement is governed by the laws of the Great State of Florida. If this Product was acquired outside the United States of America all pertinent international treaties apply.

Table Of Content

1 INTRODUCTION

1.1 Hardware Overview

The R-515JC with Reads51 (**R**igel's **E**mbedded **A**pplications **D**evelopment **S**ystem) constitutes a complete system for developing embedded control applications. Reads51 is an Integrated Development Environment (IDE) that currently supports Rigel's 8051 family embedded controller boards. Reads51 contains an assembler, SmallC-compatible C compiler, editor, monitor program, debugger and chip simulator for the 8051 family of microcontrollers.

The default configuration of the board uses our **RROS**, **R**OM **R**esident **O**perating **S**ystem and 32K of RAM. RROS has been the standard monitor for Rigel's 8051 boards. It uses the fixed Baud rate of 9600. It assumes 32K of ROM and 32K of RAM. Code and data memory banks overlap. User programs must run in 32K of code. For details on the RROS please see the separate RROS document.

In November 2000, we added another monitor option, the **RRM**, **R**AM **R**esident **M**onitor. RRM lets the user program run with 64K of code and 64K of data memory. The memory banks may also be overlapped. RRM detects the Baud rate automatically. With standard 8051 products with an 11.0592MHz crystal and using Timer 0 to generate the Baud rate, RRM supports up to 57600 Baud. RRM loads up to 64K of the user code. When RRM switches to the user code, it removes itself from the memory map. If the code and data memory banks are kept separate, 64K of code may be run with 64K of external data. For details on the RRM, please see Section 8.

The R-515JC is a six-layer board with separate VCC and Ground planes intended for demanding industrial applications. Ideal for data logging or mixed mode communications, this board is the newest in Rigel's expanding line of 8051 controllers. It's low cost and flexibility make it ideal as a training platform. The R-515JC has four serial ports, one a dedicated RS-232 port, one a synchronous serial port, and two which can be configured for RS-232 or RS-422 / RS-485. The R-515JC also has CAN, Controller Area Network with the physical layer on-board. The board comes standard with 32K RAM and 32K EPROM, but will accept up-to 512K RAM and 128K EPROM. There is a Y2K Real Time Clock / Calendar (RTCC) which may be populated as well as a battery back-up system for both the RAM and RTCC. The processor supports 10-bit A/D, as well as having 48 I/O lines, 3 16-bit timers and a watchdog timer. For more details on the processor and board, see the listings below.

1.1.1 C515C Processor Includes:

- 64k byte on-chip ROM (external program execution is possible)
- 256 byte on-chip RAM
- 2K byte of on-chip XRAM
- Up to 64K byte external data memory
- Superset of the 8051 architecture with 8 datapointers
- Up to 10 MHz external operating frequency (1 □s instruction cycle time at 6 MHz external clock)
- Eight ports : 48+1 digital I/O lines, 8 analog inputs
- On-chip Full-CAN (Controller Area Network) controller system
- Three 16-bit timer/counters
- 8 channel, 10-bit A/D converter with multiplexed inputs and built-in self calibration
- Full duplex serial interface with programmable baudrate generator (USART)
- SSC synchronous serial interface (SPI compatible)
- Seventeen interrupt vectors, at four priority levels selectable
- Extended watchdog facilities
- Power saving modes
- CPU running condition output pin
- ALE can be switched off
- Multiple separate VCC/VSS pin pairs

1.1.2 The R-515JC Board:

- 32K of SRAM / optional 128K - 512K SRAM
- 32K of monitor EPROM / optional 64K EPROM
- Optional Y2K Real-Time Clock / Calendar (RTCC) (uses DS1685)
- Optional battery back-up for RAM and Real-Time Clock / Calendar
- 3 full-duplex serial ports on board

CPU UART port with a RS-232 driver
Optional Dual UART 88C92 for two additional serial ports
Each of these ports may be used with either an RS-232 driver or an RS-422/RS-485 driver
2 Serial ports terminate at DB-9 connectors
Optional header style connectors on board for all serial ports

- CAN (Controller Area Network) physical layer (80C250 or 80C251) built into the board
- CAN port terminates on screw-type terminal blocks
- 12 general purpose digital input/output bits on screw type terminal blocks
- Two-way reset feature gives access to all interrupt vectors
- Demultiplexed processor Address and Data lines
- All system signals are available on a 96-pin header
- Power supplied to the board by way of a 2 position terminal block
- Power on LED
- Board operates on +5 volts
- Operating temperature 0 to 70C
- Machine screw sockets under all through-hole IC's
- 4 layer 4" x 5.5" board
- Mounting holes in corners

1.2 Software Overview

1.2.1 Reads51

Rigel Corporation offers 2 versions of our Reads51 software. Please select the version that will work best in your system. We recommend new users select Reads51 Version 4.1, Toolchain 4.

1. Reads51 4.x (IDE, SmallC-compatible 8051 compiler, assembler, linker, editor, chip simulator, assembly language debugger, monitor, 95/98/NT)
2. Reads51 version 2.0 (IDE, assembler, editor, debugger, monitor, DOS -- runs in Win 3.1 box)

Reads51, version 4.x, is Rigel Corporation's Integrated Development Environment for the 8051 family of processors. Reads51 constitutes a complete system for developing embedded control applications when used with Rigel Corporation's 8051 boards. Efficient software development and rapid hardware prototyping are combined in a single integrated development environment. Reads51 v4.x includes an IDE, SmallC-compatible 8051 compiler, assembler, linker, editor, chip simulator, assembly language debugger, and monitor. Reads51 v4.x is written in native 32-bit code to run on Windows95/98 and WindowsNT. Reads51 includes a sophisticated project management system to simplify code reusability and version control. Reads51 supports a full debugger in assembly language. 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.

V4 Compiler
The compiler is written to accompany Rigel's educational packages. It is SmallC compatible (integer and char only, one-dimensional arrays, one level of indirection, i.e. pointers). Please refer to books on SmallC for more information.

Running Compiler-Generated Code
The compiler is written for Rigel's 8051 family of boards. After building your project, download it to the board and swap the memory so that RAM occupies the lower block of memory.

The Reads51 software has the following distinctive features:

- Project management for organized software development
- Enhanced graphical user interface for easy monitoring
- Stand alone compiler and editor applications connected to Reads51 in a client/server fashion
- Real-Time Kernel
- RchipSim51 with SimTTY and SimI/O options.

The 8051 boards are designed to communicate with a PC (IBM PC or compatible) acting as a host. The host-to-board communications are carried out through a serial port (COM1 - COM4).

The monitor program (RROS) includes a monitor system and user-accessible system calls for control and communication support. The RROS monitor may be used to communicate with an ASCII terminal when the PC host is unavailable. The source code of the user-accessible systems calls is provided. These routines as well as all examples in the User's Guide, on the CD-ROM, and downloaded from the WEB may be used or incorporated into applications by the registered buyer without any royalties, fees, or limitations. Rigel Corporation is not responsible for the suitability or correctness of the example software. Refer to the warranty for additional information.

1.2.2 Example Software

Tutorial source code is provided to experiment with the capabilities of the R-31JP board and Reads51. Examples are designed to illustrate the features of the 8051 family of microcontrollers, specifically digital and serial input/output, timers and counters, and interrupt logic. The example software may be found in the Work directory of the Reads51 software. Please refer to the comments embedded in the programs for further information.

Users are encouraged to modify the circuit diagrams and example software in developing their own specific applications. The source code of the user-accessible systems calls, as well as all examples on the distribution disk may be used or incorporated into applications by the registered buyer without any royalties, fees, or limitations. Rigel Corporation is not responsible for the suitability or correctness of the example software. Refer to warranty for additional information.

1.3 Package List

Your R-515JC / READS package includes the following:

- R-515JC populated with 32K RAM, 32K EPROM
- R-515JC/ Reads51 User's Guide, download from web
- Reads51 software with on-line help and User's Guide, download from web
- Source code for user-accessible system included with the Reads51 software
- Example software included with the Reads51 software

A serial modem cable with a male DB9 connector and a regulated 5-volt 500mA power source are to be supplied by the user.

2 SOFTWARE SETUP

2.1 System Requirements

Reads51 Version 4.x is designed to work with an IBM PC or compatible, Pentium 120MHz or better, running Windows 95, 98, or Windows NT. The newest version of the software is always available to download off our web site, www.rigelcorp.com. We encourage you to check our web site often to keep up-to-date.

2.2 Software Installation, Reads51

If you receive a CD from Rigel, follow these steps:

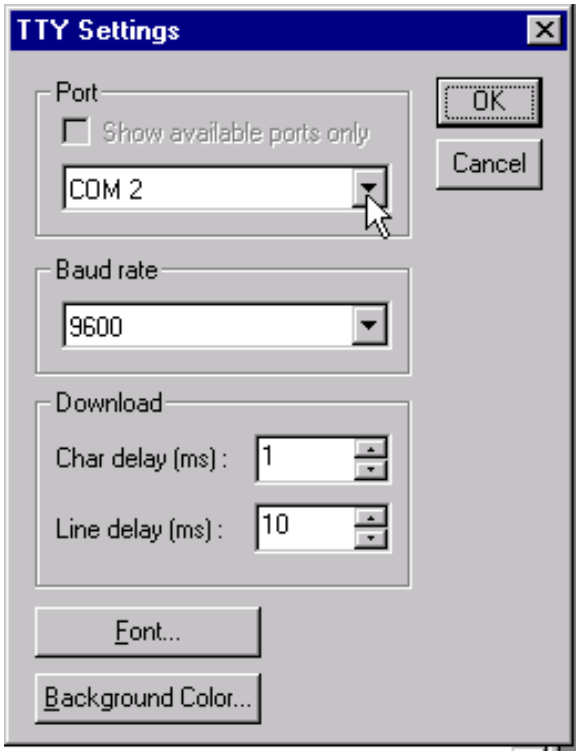
1. Place the CD-ROM in your drive.
2. Go to the Rigel Products | 8051 Software | Reads51 | Win95-nt | and click on the SetupReads400.exe file. The program will then install in your system.
3. Follow the standard install directions.

If you download the software from the web, (www.rigelcorp.com)

1. Click on the SetupReads400.exe file. The program will then install in your system.
2. Follow the standard install directions.

2.3 Quick Start

2.3.1 R-515JC Start Up



1. Check to make sure jumpers are the EA#, VPP, and PE# positions on the header.
2. Check to make sure the jumper above U6 is in the <256K position.
3. Check to make sure the slide switch is in the MON position.
4. Connect the R-515JC to the PC host via a modem cable in the P1 DB9 connector.
5. Connect the R-515JC to a well-regulated 5-Volt supply. The red LED should light up when power is connected.
6. Run the Reads51 host driver by selecting **Start | Programs | Reads51**. You may also start Reads51 by double clicking on the Reads51 short cut icon if installed.
7. Specify the serial port (COMM Port) that is connected to the board by opening the **Options | TTY Options** window.
8. Select the Toolchain and Target platform by selecting **Options | Toolchain/Target** and selecting Reads51 Toolchain v4 and the target RROS.
9. Open the TTY window using the menu command **View | TTY Window**.
10. Press RESET on the embedded controller board and observe the prompt in the TTY window.

2.3.2 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 list of monitor commands is displayed with the **H** command while the monitor program is in effect. The **H** command displays the following table.

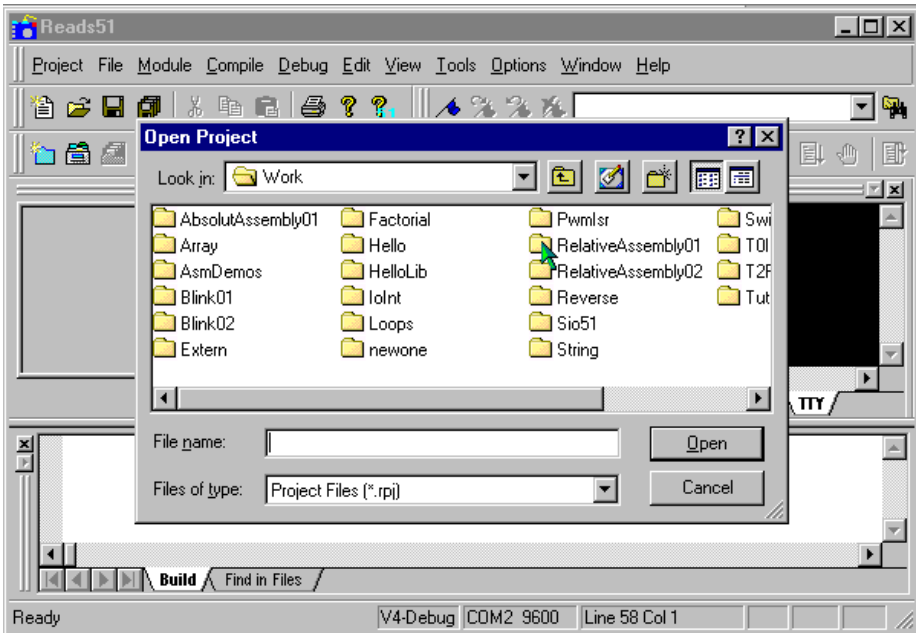
B xxxx	sets Break point at address xxxx
C xxxx-xxxx	displays Code memory
D xx-xx	displays internal Data ram
D xx=nn	modifies internal Data ram
D xx-xx=nn	fills a block of internal Data ram

G xxxx	Go - starts executing at address xxxx
H	Help - displays monitor commands
K	Kills (removes) break point
L	down Loads Intel hex file into memory
P x	displays data on Port x
P x=nn	modifies data on Port x to nn
R	displays the contents of the Registers
S	displays Special function register addresses
S xx-xx	displays Special function registers
S xx=nn	modifies Special function registers
S xx-xx=nn	fills Special function registers
X xxxx-xxxx	displays eXternal memory
X xxxx=nn	modifies eXternal memory
X xxxx-xxxx=nn	fills eXternal memory

A single-letter command may be followed by up to 3 parameters. The parameters must be entered as hexadecimal numbers. Each 'x' above represents a hexadecimal digit (characters 0..9, A..F). Intermediate spaces are ignored. Alphabetic characters are converted to upper case. The length of the command string must be 16 characters or less. The command syntax is:

Letter [address][-address][=data]<CR>.

2.3.2 Downloading and Running an Assembly Program



1. Use the use the **Project | Open Project** command to open the project AbsoluteAssembly01.
2. Assemble the program using the **Compile | Build** command.
3. Use the **Compile | Toggle Mode** command to switch to the Run/Debug Mode.
4. Click on the **Compile | Run** command and specify the starting address 8000 (hex).
5. Again the **Compile | Toggle** Mode command to revert back to the Build Mode.

2.3.3 Downloading and Running a C Program

1. Use the **Project | Open Project** command to open the project Hello.rpj in the Hello folder.
2. Open the file a01.c in the editor window by clicking on the folder in the project window.
3. Initialize the serial port for 10MHz by editing

```
// --- initialize serial port (9600 Baud) ---
 InitSerialPort0(_9600_11_059_TIMER1);
```

to read

```
// --- initialize serial port (9600 Baud) ---
 InitSerialPort0(_9600_10_C515BRG);
```

This initializes the serial port for the R-515JC board.

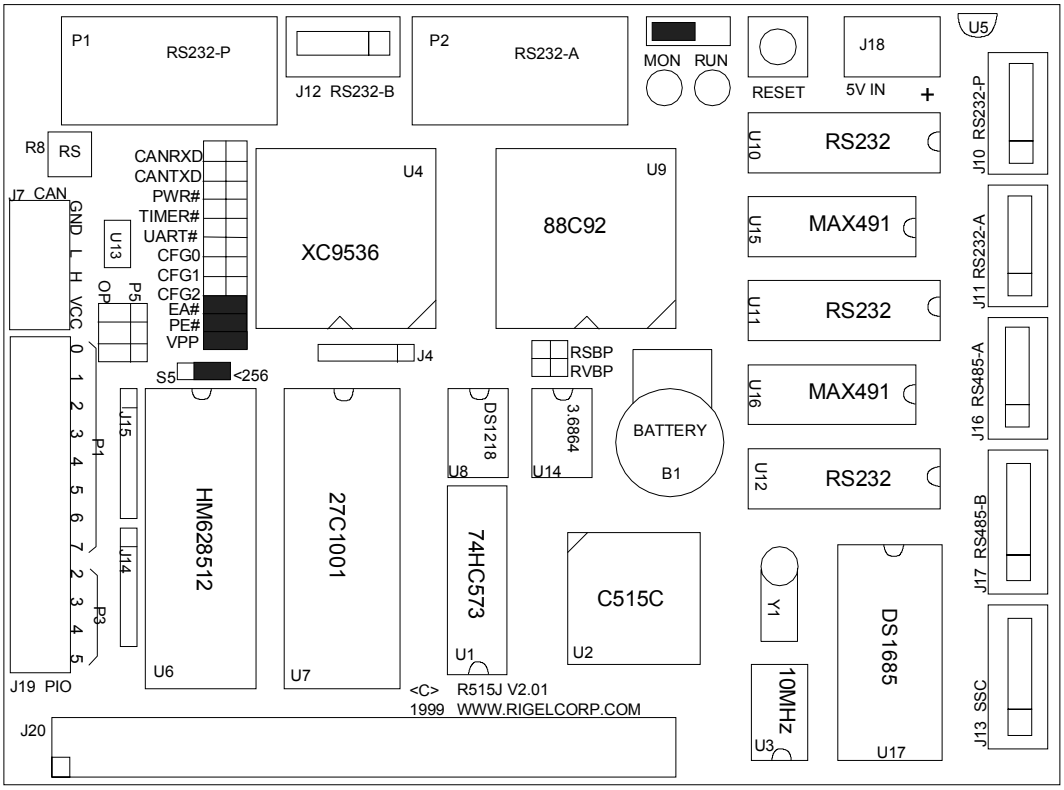
4. Compile the program and download it to the board using the **Compile | Build and Download** command. The project will be compiled and the resultant HEX code will be downloaded to the target board.

5. Press and hold the RESET button on the board. While the RESET button is pressed, flip the MON / RUN switch to the RUN position. This swaps the memory map on the board so that RAM occupies low memory. The HEX code downloaded to RAM executes when you release the RESET button.

3 OPERATING NOTES

3.1 Overview

R-515JC uses the Siemens C515C processor, an 8051 compatible microcontroller in the 80 pin MQFP package. These microcontrollers use the MCS-51 instruction set. The R-515JC uses external RAM during the development cycle. Once an application program is developed, it may be permanently placed in EPROM, or battery backed RAM. With an application-specific program installed, the R-515JC may be used as an embedded controller.



The R-515JC has 12 terminal blocks connected to Port 1 and 4 bits of Port 3. Each port may be used as either an input or an output port. The C515C ports may be operated in the 8051 mode. That is, the ports contain open-drain transistors with pull-up resistors. In this mode, when used as output ports, it is recommended that the ports sink current. Similarly, when used as input ports, first write 1's to the ports and have the external signal drive the port low. Some of the C515C ports may be configured as true push-pull ports. Refer to the C515 data book for further information. The top overlay above is not drawn to scale. It is a drawing to show position of the various components and of the default jumper and switch settings.

3.2 Power, J18

Power is brought to the R-515JC board by a two-position screw-type terminal block, J18. A well-regulated (+/- 5%) 5V DC source is required. The (+) terminal is marked on the board. Note that a diode is placed across the input in reverse. If the power is applied to the R-515JC board in reverse polarity, the diode will short the power supply attempting to prevent damage to the board. Note, the board may optionally be powered by the CAN connector (J7) which has GND and VCC located on 2 of the terminal blocks, or the 96-pin I/O header (J20) which contains posts for GND and VCC.

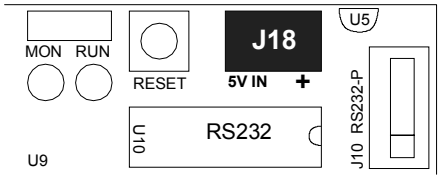


Figure 3.2 Power Connector

3.3 Communication

There are 4 serial ports and one CAN port on the R-515JC. The serial ports consist of 3 asynchronous serial ports and 1 synchronous serial port. Two of the asynchronous serial ports can be accessed by the DB9 connectors located at the top of the board or headers mounted on the right side of the board. The third serial port and the synchronous serial port are available only on the corresponding headers mounted on the right side of the board. The CAN port is available on terminal blocks located on the left side of the board.

3.3.1 Asynchronous Serial Ports

There are three asynchronous serial ports available on the R-515JC board. The DB-9 connectors P1 and P2 are only available as RS-232 serial ports and may be used to communicate with the PC host during programming and debugging the board. The serial port P1 is always populated and used as an RS-232 serial port. The other two serial ports are optional and may be used as RS-232 or RS-422/RS-485. P2 is connected to channel A of the dual UART. Note that if only one serial port is needed, the dual UART may be left unpopulated.

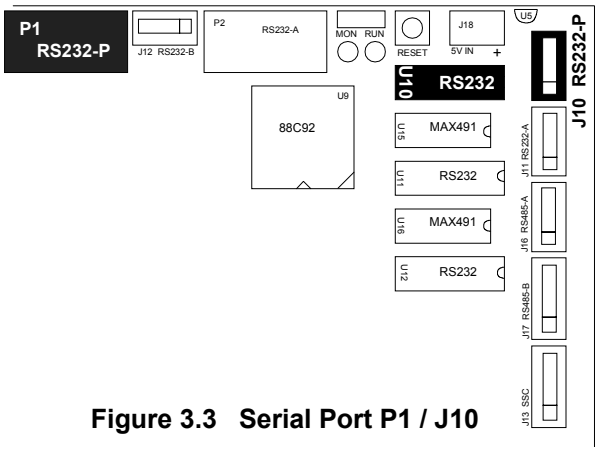


Figure 3.3 Serial Port P1 / J10

3.3.1.1 Serial Port P1, J10

The CPU serial port on the R-515JC is accessed through an RS-232 level converter in U10. The microcontroller supports the CPU transmit and receive signals. A minimal serial port is constructed with just 3 lines: transmit, receive, and ground, disregarding all hardware handshake signals. P1 of the R-515JC is a DB-9 female connector used to connect the board to an IBM compatible PC. A straight-through modem cable may be used. That is a cable connecting pin 2 of the R-515JC to pin 2 of the host, and similarly pin 3 to pin 3, and pin 5 to pin 5. P1 is also available on the header J10 labeled RS-232-P. Furthermore, P1 RS-232-level receive (SR0) and transmit (ST0) signals are available on the 96-pin I/O Header J20. The corresponding TTL-level signals from the microprocessor are also available on this header.

3.3.1.2 Using the Optional Serial Ports

There are two additional serial ports available on the R-515JC. These ports are optional. These are made available with the use of a dual UART the 88C92 which will only be populated if additional serial ports were requested at the time the board was purchased. The additional serial ports may be used as either RS-232 or RS-422 / RS-485 ports. When using the optional serial ports please note that although there are sockets on the boards for 2 RS-232 IC's (U11, and U12) and 2 sockets for the RS-485 IC's (U15 and U16) care should be taken to populate only one driver for each port. For channel A of the dual UART, either an RS-232 driver (U11) or an RS-422/ RS-48 driver (U15) may be used. Do not populate both U11 and U15. Note that the receive outputs of the to drivers terminate at the same UART pin. Since both drivers will not be receiving the same data, populating both drivers would result in a conflict at this pin. Similarly, for dual UART channel B, do not populate both U12 and U16. Refer to circuit diagrams for further connectivity information.

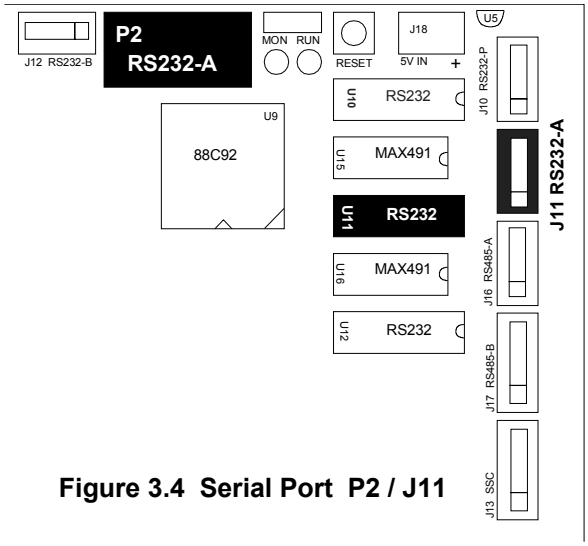


Figure 3.4 Serial Port P2 / J11

3.3.1.2.1 Serial Port, P2, RS-232-A

The DB-9 connector P2 may be used as a second RS-232 serial port for debugging when U11 is populated with an RS-232 IC. The dual UART supports the CTS and RTS hardware handshake signals. The handshake signals are not active in the default configuration. The automatic generation and use of these handshake signals are selected by programming the 88C92 control registers, usually in the initialization code. Refer to the 88C92 data book for further information on the available operating modes.

P2 includes the 5 lines: transmit, receive, CTS (Clear To Send), RTS (Request To Send) and ground. A straight-through modem cable may be used when connecting P2 to an IBM-PC serial port. That is a cable connecting pin 2 of the R-515JC to pin 2 of the host, and similarly pin 3 to pin 3, pin 5 to pin 5, pin 7 to pin 7, and pin 8 to pin 8. P2 is also available on the header J11 labeled RS-232-A. The 'A' indicates the association with channel A of the dual UART.

Pin	P2 of the R515JC	PC Serial Port
2	TXD	RXD
3	RXD	TXD
7	CTS	RTS
8	RTS	CTS
5	Ground	Ground

Note that CTS is an input to the transmitter. It is generated by the receiver at the other end of the serial link. When active, it signifies that the transmitter is clear to send data. RTS is an output from the receiver. It is sent to the transmitter at the other end of the serial link. When active, it signifies that the transmitter is requested to send data. RTS and CTS may be viewed as the two ends of the same handshake signal wire.

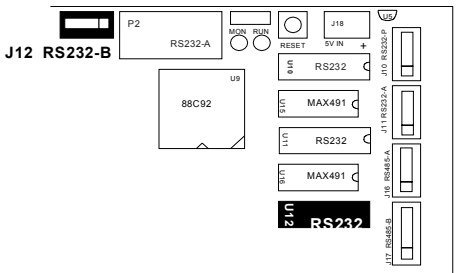


Figure 3.5 Serial Port RS-232-B

3.3.1.2.2 Serial Port, J12, RS-232-B

The TXD, RXD, CTS, RTS, and Ground signals from the dual UART Channel B RS-232 driver terminate at J12. The header J12 is located between the two DB-9s. The function and structure of J12 is similar to J11, except that J12 is associated with Channel B, whereas J11 is associated with Channel A of the dual UART.

3.3.1.2.3 Serial Port, J16, RS485-A

J16 is used for the dual UART Channel A, when Channel A is used as an RS-422 / RS-485 port. J16 is connected to the IC in socket U15. When populated this socket contains a MAX491, an RS-422 / RS-485 driver chip. Note that only one chip should be populated in the sockets used for Channel A. Do not populate both sockets U11 and U15 at the same time. Please see the Maxim Data book or web site for details on this chip.

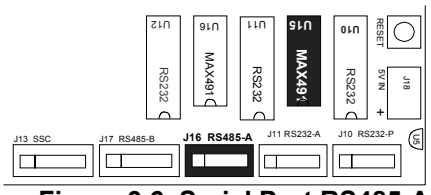


Figure 3.6 Serial Port RS485-A

3.3.1.2.4 Serial Port, J17, RS485-B

J17 is used for the dual UART Channel B when Channel B is used as an RS-422 / RS-485 port. J17 is connected to the IC in socket U16. When populated this socket contains a MAX491, an RS-422 / RS-485 driver chip. Note that only one chip should be populated in the sockets used for Channel B. Do not populate both sockets U12 and U16 at the same time. Please see the Maxim Data book or web site for details on this chip.

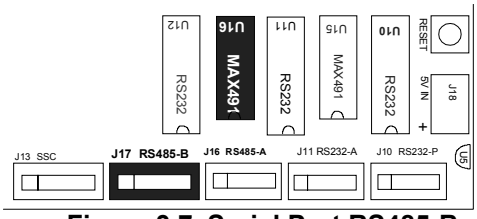


Figure 3.7 Serial Port RS485-B

3.3.2 Synchronous Serial Port

J13 is the header used for the configurable synchronous serial port on the board. Please see the Siemens data book for details on using the SSC.

3.3.3 CAN Terminal Block, J7

The CAN LO and CAN HI lines are brought to the terminal block marked J7. A two-wire twisted pair may be used to connect the R-515JC board to other CAN devices. The terminal block J7 also carries the VCC and GND lines for reference or to provide power. Please note that the CAN signals CANRS, CANLO, and CANHI are also available on the 96-pin I/O Header J20.

3.3.3.1 CAN Physical Layer Considerations

The R-515JC uses the 82C250 chip to provide a physical layer interface to a twisted pair. The outputs of the 82C250 are brought directly to J7 as CAN-LO and CAN-HI without any pull-up or termination resistors. If your physical layer contains no resistors, for example, if you connect two R-515JC boards together, pull-up (and pull-down) resistors will improve the line characteristics. The optimum resistor values depend on the transmission speed and the properties of the twisted pair. Note that the resistance and the capacitance of the twisted pair increase with the length of the connection. Also, the capacitance of the line becomes more important as the transmission speeds increase. Typically, two external resistors (1K to 10K) are sufficient, one pulling CAN-LO to VCC and the other pulling CAN-HI to GND. That is, use a pull-up resistor on CAN-LO and a pull-down resistor on CAN-HI. These resistors are in locations R9 and R10 and are not populated on the board.

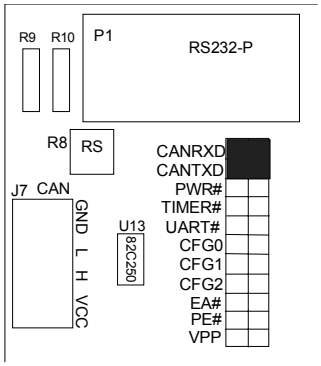


Figure 3.8 CAN Interface

3.4 Potentiometer R8

The potentiometer R8 is used by the slope control feature of the CAN interface chip, the 82C250. Turning R8 clockwise increases its value. Turned all the way counterclockwise, R8 connects the slope control pin of the 82C250 to ground. Slope control refers to desensitizing the CAN interface chip to fast acting transients. External noise sources may induce interference on the twisted pair CAN physical layer, which appear as transients to the CAN interface chip. When R8 is about zero, (grounding the slope control pin of the 82C250), the CAN interface chip processes the signals without measuring their slopes. This gives the fastest possible operation of the CAN interface chip. In this case, the physical layer should be coaxial or well insulated from external noise sources. As the value of R8 increases, the CAN interface chip enters its slope control mode. The higher the value of R8 (more counterclockwise rotation), the less sensitive the chip is to fast acting signals. This is more desirable in a noisy industrial environment when unprotected twisted pair physical media are used.

3.5 Switches

The R-515JC has one reset button and one slide switch. The pushbutton resets the board. The slide switch changes the memory configuration. The precise effect of the slide switch depends on the memory configuration (referred to as the memory mode, or simply as the mode) selected by the jumpers. Please refer to the Section 5 for detailed information about the memory modes.

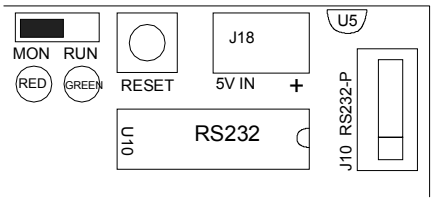


Figure 3.9 Switches and LEDs

3.6 LEDs

There are two LEDs on the R-515JC. The LEDs indicate the current operating mode of the board. They also provide visual confirmation that power is applied.

The LEDs light up depending on the operating mode which in turn is determined by the position of the slide switch. The MON LED (red) is lit when the R-515JC is run with the monitor program in the lower 32K of memory. The RUN LED (green) is lit when the board is operated with the RAM in the lower 32K of memory.

4 JUMPER SELECTION

4.1 EA#, (J1)

EA# jumper (External Enable) J1 is located in the row of headers just left of IC U4, the XC9536. Inserting a jumper in J1 grounds the EA# signal, removing the jumper connects the EA# signal to VCC through a pull-up resistor. The EA# signal selects between external memory and the on-board ROM of a processor, if present. The jumper, when installed, causes all instructions to be fetched from external memory. Since the microprocessors do not have valid code in factory-masked internal ROM, the EA# jumper is always installed in the standard configuration.

4.2 PE# (J2)

The PE# jumper (J2) is located in the row of headers just left of IC U4, the XC9536, right below the EA# jumper. The PE# jumper is inserted to ground the PE# signal. A low level (grounding) on this pin allows the software to invoke the CPU power down, idle and slow down modes. Removing the jumper at PE# holds the signal high which blocks the use of the software controlled power saving modes. If PE# is at a logic low level during reset, the watchdog timer is turned off. A high logic level during reset starts the watchdog timer. When left unconnected this pin is pulled high by an internal pull-up resistor.

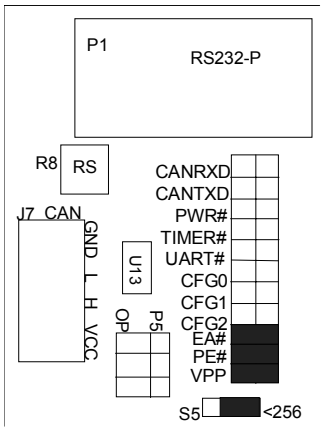


Figure 4.1 Default Jumper Settings

4.3 VPP (J6)

The VPP jumper is used when the board is populated with a 512K EPROM in the U7 socket. The EPROM in U7 contains the RROS (ROM Resident Operating System) needed for host-to-board communications. If the EPROM is 32K the VPP jumper will not be populated.

4.4 <256K (S5)

Jumper S5 is a three-pin header. A jumper must be inserted, connecting the center post to one of the outside posts. The size of the RAM placed in the socket for U6 determines the jumper position in the header. For 32K or 128K RAM devices, place the jumper in the position marked “<256K." For 512K devices, place the jumper opposite to the “<256K” mark.

4.5 PWR#/ TIMER#/UART# (J3)

J3 is actually a collection of three jumpers. One side of the jumpers are connected to P7.0 of the C515C. This port functions as an interrupt input (INT7#) of the C515. The real-time clock / calendar has two interrupt output, namely, PWR# and TIMER# (IRQ#). Similarly, the dual UART has an interrupt output UART# (INT#). All three interrupt sources are generated by open-drain-type outputs. Simply connecting the interrupt sources is equivalent to a logical OR operation. From top to bottom, the J3 jumpers connect the Timer interrupts PWR# and IRQ#, and the UART interrupt INT# to P7.0. Refer to the circuit diagrams for further information.

4.6 CFG0/CFG1/CFG2 (J5)

J5 is actually a collection of three jumpers. These jumpers are connected to the CPLD to select among the available memory modes. Refer to Section 5 for further information on the memory maps and memory modes.

4.7 CANRxD, J8 and CANTxD (J9)

The two jumpers J8 and J9 connect the CAN receive and transmit signals to the microcontroller pins P4.7 and P4.6, respectively. Remove these jumpers if P4.6 and P4.7 are to be used as general-purpose input/output port bits. Note that removing J8 and J9 prevents the microcontroller from accessing the CAN physical layer driver U13.

4.8 RVBP (J21) and RSBP (J22)

RVBP and RSBP are bypass jumpers. Populate J21 and J22 if the battery backup is not used. In this case, not only the battery but the battery backup controller chip U8 (DS1218) is removed. In battery-backed operation, the controller chip, U8, monitors the supply voltage and the memory select signal. Memory select is disallowed if the supply voltage (VCC) falls below a limit. In this case, additionally, the battery supplies power to the memory device. When VCC rises above the limit, the memory chip select signals are allowed, and the memory device is powered by VCC.

Jumper J21 is the RAM Voltage By-Pass jumper. It bypasses the controller chip to directly supply the RAM with VCC. Similarly, Jumper J22 is the RAM Select By-Pass jumper. It bypasses the controller chip to directly provide the RAM chip select signal.

4.9 Bank Address Source Selections (S2, S3, S4)

There are two ways to generate the bank addresses A16, A17, and A18. Either Port 5 bits P5.0, P5.1, and P5.2 or the dual UART general-purpose output bits OP4, OP5, and OP6 may be used. The source of each bank address is determined by the jumpers S2, S3, and S4 respectively. Note that the port bits are at logic level high after a reset. The CPLD inverts the source signal so that the bank addresses are all at logic level low (0) after a reset. This assures that code is fetched from address 0 immediately following a reset.

4.10 Default Jumper and Switch Settings

The following are the default settings for the R-515JC as sent from the factory.

- A jumper in the EA# header.
- A jumper in the PE# header.
- A jumper in the VPP header.
- A jumper in the '<256' position of S5.
- The slide switch in the 'MON' position.

The top overlay on page 7 shows the default jumper settings and switch position.

5 MEMORY CONFIGURATION

The R-515JC may take up-to 512K RAM and 128K EPROM. The board is routinely populated with 96K of memory, of which 64K is EPROM and 32K is static RAM. Program memory and external data memory are decoded to overlap. In this configuration, programs may be downloaded from the PC host, placed in RAM as data, and subsequently executed as program instructions.

5.1 Memory Addressing

The 8031 family of microcontrollers address 64K of program. The microcontroller may read from both external data memory and external code memory using the movx and movc instructions. The microcontroller may write to external data memory but may not write to external code memory.

The microcontroller pin Program Segment Enable (PSEN#) is activated (made logic 0) when a byte is to be read from external program memory, and pin Read (RD#), when a byte is to be read from external data memory. By combining these signals by an AND gate (PSEN# AND RD#), the same physical 64K memory block is made to appear as both external code and data memory. The default R-515JC configuration overlaps external data and code memory blocks by combining PSEN# and RD# in this manner. This allows downloading and running programs on the R-515JC. The pound sign (#), when used as a suffix signifies that the signal is active when low. For example, external memory is enabled when the EA# line is held at logic low (close to ground voltage).

5.2 Memory Options

5.2.1 EPROM (U7)

R-515JC has a 32-pin socket for U7. This socket holds 32K, 64K, or128K of memory sing 27C256, 27C512, or 27C1001 EPROM devices.

5.2.2 RAM (U6)

R-515JC has a 32-pin socket for U6. This socket accepts 32K, 128K, or 512K of memory using 62C256, 68C1000, or 68C4000 RAM devices.

5.3 Memory Modes

The R-515JC accommodates up to 128K of EPROM and 512K of RAM. Memory is decoded by the CPLD device, the XC9536. Memory modes refer to the way memory is decoded. More specifically, the way the physical addresses are derived from logical addresses. The physical addresses are those seen by the memory devices, and the logical addresses are those generated by the microcontroller. The default CPLD program supports three major memory modes, selected by the configuration jumpers CFG0, CFG1, and CFG2, and by the MON/RUN slide switch. The CPLD program is given in Section 7. Note that the CPLD may be reprogrammed to support a very wide range of custom memory modes, for example to relocate memory-mapped input/outputs.

Memory Mode	CFG2	CFG1	CFG0
RROS	No	No	No
NVRAM	No	See Text	Yes
Banking	Yes	X	X

The 'X's stand for "don't care," meaning the corresponding mode is active irrespective of a jumper being present at this location.

5.3.1 Memory-Mapped Input/Output

There are three memory-mapped input/output ranges corresponding to the user I/O select (XIOSEL#), the external dual UART, and the Real-Time Clock Calendar (RTCC). The default CPLD equations (As of May 2000, boards shipped before MAY 2000 use the old base address) fix the address of memory-mapped input/outputs to the following base addresses:

Peripheral	Old Base Address	New Base Address
Real-Time Clock Calendar	0xFC00	0xF400
UART	0xFD00	0xF500
XIO	0xFE00	0xF600

Note that the memory-mapped input/output ranges are effective in all memory modes. That is, no memory select is active in these ranges, irrespective of the memory mode.

The RAM and CAN are enabled by clearing bit 0 of the SYSCON register.

The C515C CAN unit uses external data memory range [0xF700..0xF7FF].
The C515C also contains 2K on-chip RAM mapped to external data memory range [0xF800..0xFFFF].
Jumpers CFG0, CFG1, and CFG2 are removed for the RROS mode. This is the default memory mode used with the Reads51 environment. It is compatible with the earlier Rigel boards. It uses 32K of ROM (EPROM) and 32K of RAM. ROM and RAM are overlapped, so that user programs may be downloaded as data and executed as code. When the MON/RUN switch is in the "MON" or monitor position, ROM occupies the lower half of the 64K address range. In the "RUN" position, RAM occupies the lower half of the 64K address range.

The slide switch is used when downloaded programs need direct access to the interrupt vectors located in low memory. Such a program, with its origin at 0, is first downloaded into a RAM device placed in U6. While holding the RESET button down, S1 is moved to its RUN position. Now the program in U6 is in the low memory block, starting at address 0. Releasing the RESET button executes the user program in U6. The user program may then have direct access to all interrupt vectors.
The RAM and CAN are enabled by clearing bit 0 of the SYSCON register.
The C515C CAN unit uses external data memory range [0xF700..0xF7FF].
The C515C also contains 2K on-chip RAM mapped to external data memory range [0xF800..0xFFFF].

5.3.1.1 The RROS Memory Mode

Jumpers CFG0, CFG1, and CFG2 are removed for the RROS mode. This is the default memory mode used with the READS51 environment. It is compatible with the earlier Rigel boards. It uses 32K of ROM (EPROM) and 32K of RAM. ROM and RAM are overlapped, so that user programs may be downloaded as data and executed as code. When the MON/RUN switch is in the "MON" or monitor position, ROM occupies the lower half of the 64K address range. In the "RUN" position, RAM occupies the lower half of the 64K address range. The slide switch is used when downloaded programs need direct access to the interrupt vectors located in low memory. Such a program, with its origin at 0, is first downloaded into a RAM device placed in U6. While holding the RESET button down, S1 is moved to its RUN position. Now the program in U6 is in the low memory block, starting at address 0. Releasing the RESET button executes the user program in U6. The user program maythen have direct access to all interrupt vectors.

5.3.1.2 The NVRAM Memory Mode

This mode is used for storing the user program in battery-backed non-volatile RAM. It assumes non-overlapped 64K of code memory and 64K of data memory. Code is placed into RAM when the MON/RUN switch is in the "MON" or monitor position. In the "RUN" position, the lowest 64K block of RAM is accessed as code memory and the next 64K block of RAM as data memory.

Slide Switch	Code Memory	Data Memory
MON	EPROM [0-FFFF]	RAM [0-FFFF]
RUN	RAM [0-FFFF]	RAM [10000-1FFFF]

The CFG1 jumper is used to toggle the A15 line so that the Reads51 environment may be used to download upto 32K of user code into non-volatile RAM. Remove CFG1 if you are using Reads51 to download user code.

5.3.1.3 The Banking Memory Mode

This mode supports code and data memory blocks larger than the 64K addressed by the microcontroller. Since the EPROM may be upto 128K, one additional address bit is needed (A16). Similarly, since the RAM may be upto 512K, three additional address bits are needed (A16, A17, and A18). These bits are not automatically generated by the processor, but must be asserted by other means, such as processor ports. Changing the high address bits not generated by the processor is often referred to as bank switching. Once the proper bank is switched into, the processor generates the address lines automatically.

There are two ways to generate the bank addresses A16, A17, and A18. Either Port 5 bits P5.0, P5.1, and P5.2 or the dual UART general-purpose output bits OP4, OP5, and OP6 may be used. The source of each bank address is determined by the jumpers S2, S3, and S4 respectively. Note that the port bits are at logic level high after a reset. The CPLD inverts the source signal so that the bank addresses are all at logic level low (0) after a reset. This assures that code is fetched from address 0 immediately following a reset.

6 HEADERS

The R-515JC board has ten headers: the I/O (input/output) header J20, 6 serial port headers J10, J11, J12, J13, J16, and J17, 2 headers for the UART input and output options, J14 and J15, and J4 the header for the JTAG programming of the CPLD.

6.1 J20 (I/O Header)

The I/O header contains Ports 1, 3, 4, 5, and 6 along with the HWPD#, Reset pins and CAN transmit and receive signals. The I/O header is a three-row 96-pin header. The first 25 pins of the first two rows are compatible with our I/O boards to allow for easy access to prototyping. Individual signals of these jumpers are listed below. Pin 1 may be identified as the post with the square pad on the printed circuit board.

Pin	Signal	Pin	Signal	Pin	Signal
1	GND	**2**	VCC	**3**	GND
4	GND	**5**	VCC	**6**	D0
7	P6.0	**8**	P6.1	**9**	D1
10	P6.2	**11**	P6.3	**12**	D2
13	P6.4	**14**	P6.5	**15**	D3
16	P6.6	**17**	P6.7	**18**	D4
19	HWPD#	**20**	CANRS	**21**	D5
22	VAGND *	**23**	CANHI	**24**	D6
25	VAREF *	**26**	CANLO	**27**	D7
28	P1.0	**29**	P4.O	**30**	GND
31	P1.1	**32**	P4.1	**33**	A0
34	P1.2	**35**	P4.2	**36**	A1
37	P1.3	**38**	P4.3	**39**	A2
40	P1.4	**42**	P4.4	**42**	A3
43	P1.5	**44**	P4.5	**45**	A4
46	P1.6	**47**	P4.6	**48**	A5
49	P1.7	**50**	P4.7	**51**	A6
52	P3.2	**53**	P5.0	**54**	A7
55	P3.3	**56**	P5.1	**57**	A8
58	P3.4	**59**	P5.2	**60**	A9
61	P3.5	**62**	P5.3	**63**	A10
64	P3.0	**65**	P5.4	**66**	A11
67	P3.1	**68**	P5.5	**69**	A12
70	P3.6	**71**	P5.6	**72**	A13
73	P3.7	**74**	P5.7	**75**	A14
76	RSTIN#	**77**	P7.0	**78**	A15
79	RSTOUT#	**80**	SQW	**81**	A16
82	RSTOUT	**83**	CPUR#	**84**	A17
85	GND	**86**	XIOSEL#	**87**	A18
88	SR0	**89**	ALE	**90**	P3.6
91	ST0	**92**	PSEN#	**93**	P3.7
94	VCC	**95**	GND	**96**	GND

* The schematics use the designation P68 for VAGND and P69 for VAREF

6.2 Serial Port Headers

There are four serial ports on the board, all of which have headers for access. Of the 4 available serial ports, three are asynchronous serial ports. One of the asynchronous ports is a dedicated RS-232 port, the other two can be configured as RS-232 or RS-422 / RS-485 ports. The fourth serial port is a synchronous serial port. There are six headers to support the various serial port options.

6.2.1 J10, RS-232-P

J10 is the secondary connector for the primary serial port on the board. Pin one is designated on the printed circuit board with a square pad. See the chart below for the signals found on this header.

6.2.2 J11, RS-232-A

J11 is the secondary connector for the UART Channel A when used as a RS-232 port. Pin one is designated on the printed circuit board with a square pad. See the chart below for the signals found on this header.

6.2.3 J12, RS-232-B

J12 is the connector for the dual UART Channel B when used as a RS-232 port. Pin one is designated on the printed circuit board with a square pad. See the chart below for the signals found on this header

6.2.4 J16, RS-485-A

J16 is the connector for the dual UART Channel A when RS-422 / RS-485. Pin one is designated on the printed circuit board with a square pad. See the chart below for the signals found on this header

6.2.5 J17, RS-485-B

J17 is the connector for the dual UART Channel B when using the RS-422 / RS-485 port. Pin one is designated on the printed circuit board with a square pad. See the chart below for the signals found on this header

6.2.6 J13, SSC

J13 is the connector for the synchronous serial channel available on the board. Pin one is designated on the printed circuit board with a square pad. See the chart below for the signals found on this header.

PIN #	HEADER					
	J1O	**J11**	**J12**	**J13**	**J16**	**J17**
1	RXD	RXD	RXD	VCC	VCC	VCC
2	TXD	TXD	TXD	SR1	A	A
3	GND	GND	GND	ST0	B	B
4	XRST	RTS	RTS	SCLK	Z	Z
5	NC	CTS	CTS	SLS#	Y	Y
6	--	--	--	GND	GND	GND

6.3 J14, J15, UART Auxiliary I/O

The dual UART has 6 auxiliary inputs and 7 auxiliary outputs. Some of these are used for hardware handshaking (RTS and CTS) if so programmed. Also, three of the outputs may be used to generate the high address bits for bank switching. These inputs and outputs are also brought to two headers J14 and J15 for general-purpose use or testing. Please refer to the dual UART data book for details on programming the inputs and outputs. The pinouts for the headers are in the chart below. The square pad on the board designates pin one.

PIN #	HEADER	
	J14, U-IN	**J15, U-OUT**
1	VCC	VCC
2	IP2	OP2
3	IP3	OP3
4	IP4	OP4
5	IP5	OP5
6	IP6	OP6
7	GND	OP7
8	---	GND

6.4 J4, JTAG

The R-515JC board uses the Xilinx XC95C36 CPLD to decode memory maps. The header marked J4 on the board is the JTAG header used for programming the CPLD.

7 CPLD EQUATIONS

module r515jc
Title 'r515jc'

r515jc device;

Declarations

```
// --- inputs ---
A8         pin  8;
A9         pin  9;
A10        pin 11;
A11        pin 12;
A12        pin 13;
A13        pin 14;
A14        pin 18;
PA15       pin 19;
BA16X      pin  2;
BA17X      pin  3;
BA18X      pin  4;
PSENX      pin  7;
RDX        pin  5;
WRX        pin  6;
RSTINX     pin 34;
MON        pin 27;
AUX0       pin 20;
AUX1       pin 22;
AUX2       pin 24;

// --- outputs ---
ROMSELX    pin  1 istype 'com, pos';
ROMRDX     pin 44 istype 'com, pos';
RAMSELX    pin 43 istype 'com, pos';
RAMRDX     pin 42 istype 'com, pos';
XIOSELX    pin 26;
UARTSELX   pin 40;
TIMERSELX  pin 39;
MA15       pin 38;
MA16       pin 37;
MA17       pin 36;
MA18       pin 35;
RSTOUT     pin 33;
RSTOUTX    pin 25;
LEDMONX    pin 28 istype 'com, pos';
LEDRUNX    pin 29 istype 'com, pos';

// --- nodes ---
ARMMODE  NODE istype 'reg, buffer';
MONMODE  NODE istype 'reg, buffer';

Equations
 RSTOUTX   = RSTINX;
 RSTOUT    = !RSTINX;
 XIOSELX   = !(PA15 & A14 & A13 & A12 & A11 & A10 &  A9 & !A8); // 0xFExx
 UARTSELX  = !(PA15 & A14 & A13 & A12 & A11 & A10 & !A9 &  A8); // 0xFDxx
 TIMERSELX = !(PA15 & A14 & A13 & A12 & A11 & A10 & !A9 & !A8); // 0xFCxx
// XSELX     = XIOSELX & UARTSELX & TIMERSELX;

 when(AUX2 & !AUX0) then
  {
"   ARMMODE.AR   = 1;              // flip-flop asynchronous reset
"   MONMODE.AR   = 1;              // flip-flop asynchronous reset
```

```
  // --- NVRAM mode ---
   when(!MON) then // MON mode
    {
     LEDMONX =  0;
     LEDRUNX =  1;
     when(!PSENX & XIOSELX & UARTSELX & TIMERSELX) then // EPROM access
      {
       MA15    =  PA15;
       MA16    =  0;
       ROMSELX =  0;
       ROMRDX  =  PSENX;
       RAMSELX =  1;
       RAMRDX  =  1;
       }
      else when( (!RDX # !WRX) & XIOSELX & UARTSELX & TIMERSELX) then // RAM access
       {
//        MA15    = !PA15;   // RROS inverts A15
        when(AUX1) then MA15=!PA15; else MA15=PA15;
        MA16    =  0;
        ROMSELX =  1;
        ROMRDX  =  1;
        RAMSELX =  0;
        RAMRDX  =  RDX;
        }
       else // indeterminant
        {
         MA15    =  0;
         MA16    =  0;
         ROMSELX =  1;
         ROMRDX  =  1;
         RAMSELX =  1;
         RAMRDX  =  1;
         }
     }
    else  // RUN mode
     {
      LEDMONX =  1;
      LEDRUNX =  0;
      MA15    =  PA15;
      ROMSELX =  1;
      ROMRDX  =  1;
      when(!PSENX & XIOSELX & UARTSELX & TIMERSELX) then // low RAM access
       {
        MA16    =  0;
        RAMSELX =  0;
        RAMRDX  =  PSENX;
        }
       else when( (!RDX # !WRX) & XIOSELX & UARTSELX & TIMERSELX) then // RAM access
        {
         MA16    =  1;
         RAMSELX =  0;
         RAMRDX  =  RDX;
         }
        else  // indeterminant
         {
          MA16    =  0;
          RAMSELX =  1;
          RAMRDX  =  1;
          }
     }
  // --- end NVRAM mode ---
  }
 else when(!AUX2) then
```

```
  {
"   ARMMODE.AR   = 1;              // flip-flop asynchronous reset
"   MONMODE.AR   = 1;              // flip-flop asynchronous reset
   // --- Bank Switching mode ---
   LEDMONX =  0;
   LEDRUNX =  0;
   MA15    =  PA15;
   MA16    =  !BA16X;
   MA17    =  !BA17X;
   MA18    =  !BA18X;
   when(!XIOSELX # !UARTSELX # !TIMERSELX) then // memory mapped IO select
    {
     ROMSELX =  1;
     ROMRDX  =  1;
     RAMSELX =  1;
     RAMRDX  =  1;
     }
    else
     {
      ROMSELX =  PSENX;
      ROMRDX  =  PSENX;
      RAMSELX =  RDX & WRX;
      RAMRDX  =  RDX;
      }
   // --- end Bank Switching mode ---
   }
 else when(AUX2 & !AUX1) then
  {
  // --- XEVA mode ---
  // MONMODE=1 : code memory is in EPROM, data memory is RAM
  // RUNMODE=1 : overlapped code and data memory is 64K of RAM
  // reset                    -> MONMODE=1, ARMMODE=0
  // (RD#=PSEN#=0 and A14=1) -> ARMMODE
  // ARMMODE and PSEN#=A14=0 -> RUNMODE

  MA16   =  !BA16X;
  MA17   =  !BA17X;
  MA18   =  !BA18X;

  ARMMODE.D    =  RSTINX;              // flip-flop data input
  ARMMODE.AR   = !RSTINX;              // flip-flop asynchronous reset
  ARMMODE.CLK  = !RDX & !PSENX & A14;  // flip-flop clock

  MONMODE.D    = !ARMMODE;             // flip-flop data input
  MONMODE.AP   = !RSTINX;              // flip-flop asynchronous preset
  MONMODE.CLK  = !PSENX & !A14;        // flip-flop clock

  LEDRUNX  = !ARMMODE.Q;               // ARMMODE.Q is the flip-flop output
  LEDMONX  = !MONMODE.Q;               // MONMODE.Q is the flip-flop output

  when(MONMODE.Q) then  // XEVA MON mode
//  when(!MON) then
   {
    LEDRUNX  = 1;
    LEDMONX  = 0;
    when(!PSENX) then // EPROM access
     {
      MA15    =  1; //PA15; Using 27C512's -- upper 32K is Tele51
      ROMSELX =  !XIOSELX # !UARTSELX # !TIMERSELX;
      ROMRDX  =  0;
      RAMSELX =  1;
      RAMRDX  =  1;
      }
```

```
     else // when(!RDX # !WRX) then // RAM access
      {
       MA15    =  PA15;
       ROMSELX =  1;
       ROMRDX  =  1;
       RAMSELX =  (RDX & WRX) # !XIOSELX # !UARTSELX # !TIMERSELX;
       RAMRDX  =  RDX;
       }
    }
   else  // (MONMODE.Q==0) ... XEVA RUN mode
    {
     LEDRUNX  = 0;
     LEDMONX  = 1;
     MA15    =  PA15;
     ROMSELX =  1;
     ROMRDX  =  1;
     RAMSELX =  (PSENX & RDX & WRX) # !XIOSELX # !UARTSELX # !TIMERSELX;
     RAMRDX  =  PSENX & RDX;
     }
  // --- end XEVA mode ---
  }
 else
  {
  // --- RROS mode ---
"   ARMMODE.AR   = 1;              // flip-flop asynchronous reset
"   MONMODE.AR   = 1;              // flip-flop asynchronous reset

   MA15      = 0;
   MA16      = 0;
   MA17      = 0;
   MA18      = 0;

   when(!MON) then
    {
     LEDMONX  = 0;
     LEDRUNX  = 1;

     ROMSELX  =  PA15 # !(XIOSELX & UARTSELX & TIMERSELX);
     ROMRDX   =  RDX & PSENX;
     RAMSELX  = !PA15 # !(XIOSELX & UARTSELX & TIMERSELX);
     RAMRDX   =  RDX & PSENX;
     }
    else
     {
      LEDMONX  = 1;
      LEDRUNX  = 0;

      ROMSELX  = !PA15 # !(XIOSELX & UARTSELX & TIMERSELX);
      ROMRDX   =  RDX & PSENX;
      RAMSELX  =  PA15 # !(XIOSELX & UARTSELX & TIMERSELX);
      RAMRDX   =  RDX & PSENX;
      }
  // --- end RROS mode ---
  }
```

end r515jc

8. RRM

The RRM monitor, which was originally used with Rigel's customer-specific OEM products, is now supported by the Reads51 IDE. Newer versions of the R31JP and the R515JC support the RRM mode. RRM has two advantages over RROS: it supports higher Baud rates, and larger user programs. RRM is useful in downloading and running larger C programs on the Rigel boards.

8.1 RRM versus RROS

RROS has been the standard monitor for Rigel's 8051 boards. It uses the fixed Baud rate of 9600. It assumes 32K of ROM and 32K of RAM. Code and data memory banks overlap. User programs must run in 32K of code. As of November 2000, R-515JC's have been shipping with RRM support. R-515JC's that use a CPLD (U3 : XC9536) may be updated to support the RRM mode.

RRM lets the user program run with 64K of code and 64K of data memory. The memory banks may also be overlapped. RRM detects the Baud rate automatically. After reset, you must hit the space bar (or the enter key) to send a character to RRM. RRM deduces the Baud rate from the character and responds with its prompt. With standard 8151 products with an 11.0592MHz crystal and using Timer 0 to generate the Baud rate, RRM supports up to 57600 Baud. RRM loads up to 64K of the user code. When RRM switches to the user code, it removes itself from the memory map. If the code and data memory banks are kept separate, 64K of code may be run with 64K of external data.

8.2 Selecting RRM Mode

Use the **|Toolchain/Target|** menu under the options menu. Select RRM.

Selecting the RRM mode on the R-515JC.
Insert jumper AUX0 to activate the RRM mode with code and data memory banks separate. If you would like the code and memory banks to overlap, also insert jumper AUX2.

8.3 An Example for the R-515JC

1. Select the RRM as the target. Use the **Options | Toolchain/Target Options** menu.

2. Use the **Options | TTY Options** menu to change the Baud rate to 57600. Open the TTY window.

3. Make sure you have the jumpers AUX0 and AUX2 on the board inserted. The EA# jumper should also be inserted.

4. Move the slide switch on the board to the MON position and press the reset button. Observe that the red LED is on.

5. Click the mouse inside the TTY window and activate the caret (the vertical bar that shows the current place). Press the space bar and observe the RRM prompt. RRM is now locked to the Baud rate (576000).

6. Compile and download the C code given below. You may simply press F2 to toggle the build / debug mode, or use the menu item **Compile | Toggle Build/Debug Mode**. Open the TTY window if not already open.

7. Run the program using the **Debug | Run Skip Breakpoints** menu command (the red exclamation toolbar button also issues this command). Note the message "Hello World" in the TTY window. Also note that the red LED is off and that the green LED is on.

8. You may rerun the code without downloading it. Simply move the slide switch to the RUN position (closer to the green LED) and press RESET. This resets the R-515JC to run the user program just downloaded. If you want to return to the monitor, move the slide switch to the MON position and press RESET. Note that the red LED is now on. Every time you reset RRM, you must send a space character to the TTY window for RRM to detect the Baud rate.

```
// The "Hello World" program for 57600 Baud RRM mode for the R31JP.

#include <sfr51.h>
#include <sio51.h>

main(){
```

```
// start serial port at 57600 Baud
 TMOD=0x20;    // set counter 1 for auto reload - mode 2
 TCON=0x41;    // run counter 1 and set edge trig ints
 PCON |= 0x80; // double baud rate
 TH1=0xFF;     // set reload register (57600 baud with xtal=11.0592mhz)
 SCON=0x50;    // set serial control reg for 8 bit data and mode 1

 printf("hello world...\n\n");

 while(1); // absorbing loop
}
```

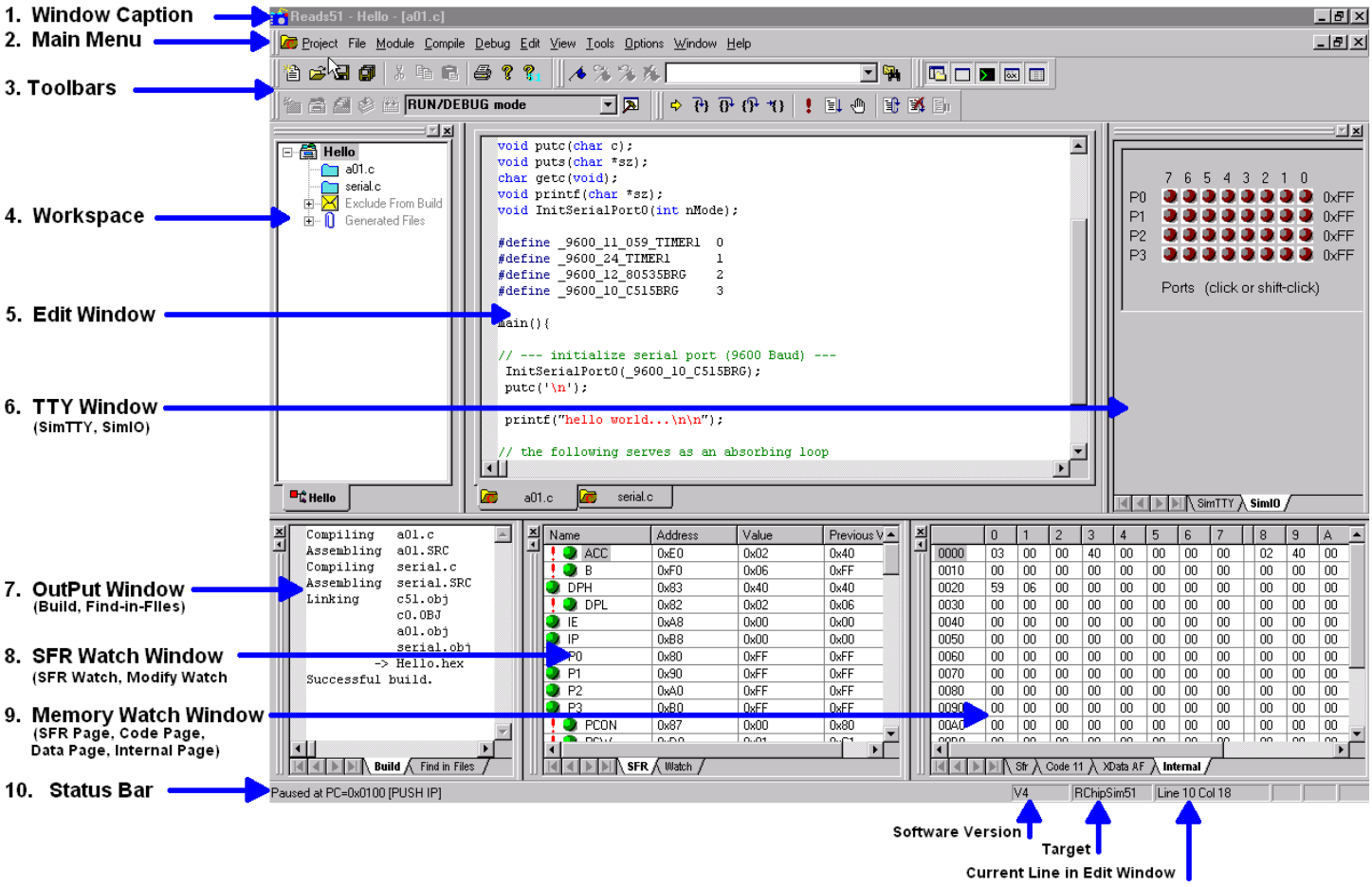
9 READS51 OVERVIEW

Reads51 is an Integrated Development Environment (IDE) that currently supports Rigel's 8051 family of embedded control boards. The IDE includes an assembler, C compiler, editor, linker/locator, debugger, and chip simulator. We've added two new features to the Reads51 software. The first is the ability to look at the Preprocessor output. To generate a file from the preprocessor go to **Tools | Run Preprocessor** and select the file you would like to look at. A **.xc file will be generated which shows the code the preprocessor has generated for the selected file. The second new feature is the ability to implement a simple real-time multithreading kernel. The Project `_rtmt` generates the library `_rtmt.lib`, found in the ".\include" directory. Projects `mt01` to `mt04`, found in the ".Pi51ca\ch05 directory, illustrate the use of the library. For more details on the use of the real- time multithreading kernel, please refer to the book "Programming and Interfacing the 8051 in C and Assembly".

Graphically, the IDE consists of the main menu, customizable toolbars, and various windows. All windows, except the editor window are dockable. Dockable windows may be attached to any side of the IDE, or left floating anywhere on the desktop.

The following list of IDE features corresponds to the comments on the diagram given on the next page.

1. Window Caption	Shows the current active project, and file.
2. Main Menu Commands	Contains the highest level menu commands
3. Toolbars	Displays a set of icons at the top of the editor window. These are shortcuts to the more often used menu commands.
4. Workspace	Shows all open projects in tabs, the active project's tab is highlighted in red.
5. Edit Window	Source modules and files open in this window for editing.
6. TTY Window	PC to Board communications shown here.
7. Output Window	Shows the result of various processes. The Build tab shows the compiler or assembler results. The Find-in-File tab reports the results of searches.
8. SFR Watch Window	Shows the value of the SFR's while debugging.
9. Memory Watch Window	Shows the value in the different types of memory while debugging.
10. Status Bar	Shows the result of various operations, software version, target, and the position of the cursor.


1. Window Caption
2. Main Menu
3. Toolbars
4. Workspace
5. Edit Window
6. TTY Window
(SimTTY, SimIO)
7. OutPut Window
(Build, Find-in-Files)
8. SFR Watch Window
(SFR Watch, Modify Watch
9. Memory Watch Window
(SFR Page, Code Page,
Data Page, Internal Page)
10. Status Bar
Reads51 - Hello - [a01.c]
Project File Module Compile Debug Edit View Tools Options Window Help
RUN/DEBUG mode
Hello
a01.c
serial.c
Exclude From Build
Generated Files
void putc(char c);
void puts(char *sz);
char getc(void);
void printf(char *sz);
void InitSerialPort0(int nMode);
#define _9600_11_059_TIMER1 0
#define _9600_24_TIMER1 1
#define _9600_12_80535BRG 2
#define _9600_10_C515BRG 3
main(){
// --- initialize serial port (9600 Baud) ---
InitSerialPort0(_9600_10_C515BRG);
putc('\n');
printf("hello world...\n\n");
// the following serves as an absorbing loop
7 6 5 4 3 2 1 0
P0 0xFF
P1 0xFF
P2 0xFF
P3 0xFF
Ports (click or shift-click)
SimTTY SimIO
Compiling a01.c
Assembling a01.SRC
Compiling serial.c
Assembling serial.SRC
Linking c51.obj
c0.OBJ
a01.obj
serial.obj
-> Hello.hex
Successful build.
Build Find in Files
Name Address Value Previous V
ACC 0xE0 0x02 0x40
B 0xF0 0x06 0xFF
DPH 0x83 0x40 0x40
DPL 0x82 0x02 0x06
IE 0xA8 0x00 0x00
IP 0xB8 0x00 0x00
P0 0x80 0xFF 0xFF
P1 0x90 0xFF 0xFF
P2 0xA0 0xFF 0xFF
P3 0xB0 0xFF 0xFF
PCON 0x87 0x00 0x80
SFR Watch
0 1 2 3 4 5 6 7 8 9 A
0000 03 00 00 40 00 00 00 00 02 40 00
0010 00 00 00 00 00 00 00 00 00 00 00
0020 59 06 00 00 00 00 00 00 00 00 00
0030 00 00 00 00 00 00 00 00 00 00 00
0040 00 00 00 00 00 00 00 00 00 00 00
0050 00 00 00 00 00 00 00 00 00 00 00
0060 00 00 00 00 00 00 00 00 00 00 00
0070 00 00 00 00 00 00 00 00 00 00 00
0080 00 00 00 00 00 00 00 00 00 00 00
0090 00 00 00 00 00 00 00 00 00 00 00
Sfr Code 11 XData AF Internal
Paused at PC=0x0100 [PUSH IP]
V4
RChipSim51
Line 10 Col 18
Software Version
Target
Current Line in Edit Window

10 READS51 CONCEPTS

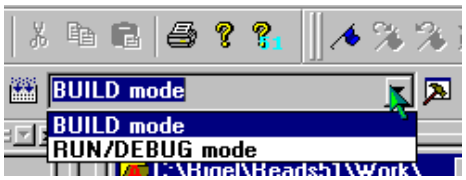
Reads51 has two modes, referred to as the "Build Mode" and the "Run/Debug Mode". The IDE Modes reinforce the typical aspects of code creation and development versus code execution and debugging. For example, the Run/Debug mode disables code editing as well as adding or removing modules while the code is being executed.

10.1 IDE Modes

Build Mode- supports source code creation and revision. All project, module, and edit functions are enabled. You may create new projects, new modules, add or remove modules, etc.

Run/Debug- is oriented to facilitate code execution and debugging. Project management and source code editing functions are disabled. The commands to run, single step, set/clear breakpoints, watch variables are enabled only in this mode.

The current mode is always displayed in the drop-down list box in the toolbar. There are four alternative actions to toggle the mode.



1. Use menu item **Compile | Toggle BUILD/DEBUG Mode**.
2. Use the toolbar button. 
3. Use the hot key F2.
4. Use the drop-down list box in the toolbar.

10.2 Projects and Modules

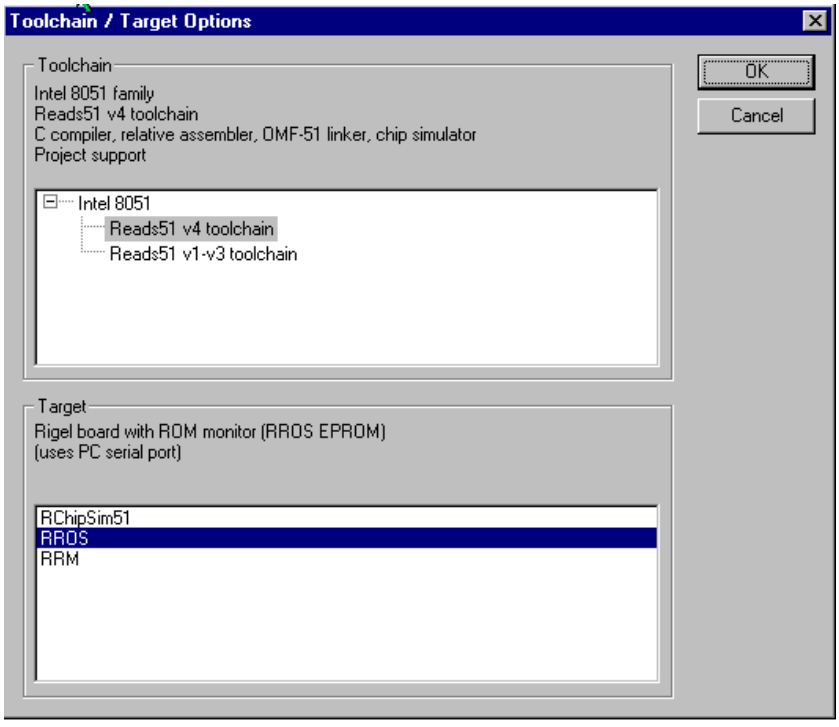
Reads51 uses a project-oriented code development and management system. Projects contain modules, which may be written in either C or assembly. Modules may freely be shared or copied from one project to another. Moving modules between projects is accomplished by the “cut”, “copy”, and “paste” commands under the Module menu or by the “Import Module” command under the Module menu.

10.2.1 Projects

A project is a collection of files managed together. Each code module in a project corresponds to a separate project file. By default 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. You may use the long names provided by the 32-bit Windows operating systems to keep different versions of your software in a controlled manner. For example, the project “Motor Control 07-20-2000” may be saved under the new name “Motor Control 07-25-2000” as new features are added. This way you may revert to an older version, if needed.

10.2.2 Modules

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 previously developed modules from an existing project to a new project by cutting and pasting or by importing. You may also add modules to a project by “drag-and-dropping” them from the Explorer Window. 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.



10.3 Workspaces

The Reads51 IDE allows multiple projects to be open concurrently. The collection of the various visual components of the projects constitute a workspace. You may save workspaces and re-open them later. When a workspace is opened, all projects

and their various components are restored. If multiple projects are open you may toggle between the workspaces by selecting the tabs at the bottom of the workspace window.

The Project menu contains three commands “Open Workspace”, “Save Workspace”, and “Close Workspace”, as well as the command “Recent Workspaces” to view or open recently saved workspaces.

10.4 Toolchains

A toolchain refers to a set of software development programs such as a compiler, assembler, and a linker, intended to be used together to perform the steps in generating executable code from various source files.

Reads51 currently contains two toolchains, v4 and v3. V3 contains the Reads51v3.x absolute assembler. The V4 contains the new (v4.00) relative assembler and linker. The v4 toolchain also includes a SmallC compatible C compiler. Use the “Options | Toolchain / Target Options” menu item to select the toolchain and target to be used. If you would like to program in C, you must select the Reads51v4 toolchain. We recommend that you use the v4 toolchain for all new projects.

We have three targets now available, the RChipSim51, the RROS, and the RRM. The RChipSim is our simulator. The RROS and RRM modes use the serial port to download code to the boards. We use the RROS, **R**OM **R**esident **O**perating **S**ystem, on all of our 8051 boards and it is the default monitor. The RRM, **R**AM **R**esident **M**onitor, was previously used only for Rigel’s custom OEM hardware. Newer versions of the R31JP and the R515JC support the RRM mode. RRM has two advantages over RROS: it supports higher Baud rates, and larger user programs. RRM is useful in downloading and running larger C programs on the Rigel boards. Check your board hardware manual to see if it supports RRM.

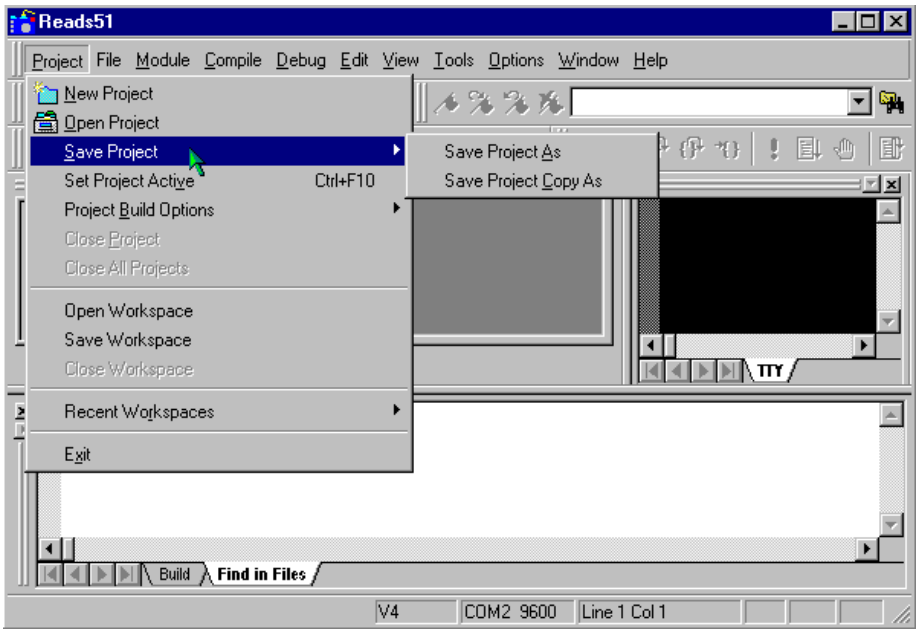
Rigel’s 8051-chip simulator is supported by both toolchains. Reads51 toolchain options are organized for future expansion of the toolchain selections, and microcontroller families. Currently, the IDE only supports the 8051 family.

11 READS51 IDE

Graphically, the IDE consists of the main menu, customizable toolbars, and various windows. All windows, except the editor window are dockable. Dockable windows may be attached to any side of the IDE, or left floating anywhere on the desktop.

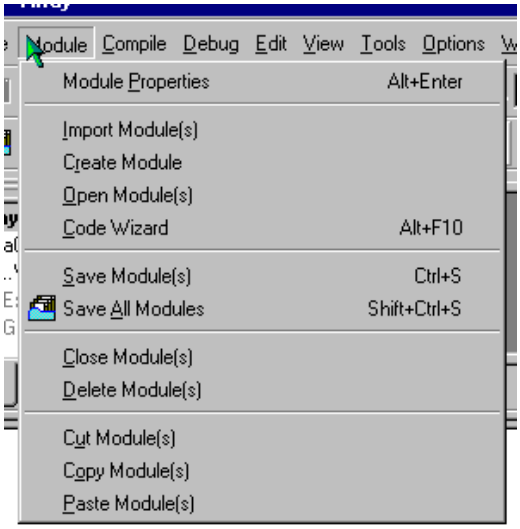
11.1 Menu Commands

The functionality of the Reads51 components remains fully integrated. The user interface has been improved by placing many of the specific commands into sub-menus. The Main Menu contains the higher-level options such as projects, modules, or tools. Most Windows also support specific pop-up menus, activated by right-clicking the mouse. For details on the menu commands see the appendix.



11.1.1 Project

Under the "Project" menu, you will find many of the familiar file commands such as, "New", "Open", "Save", and "Close". You'll also find commands, which involve the workspace and compile options.



11.1.2 File

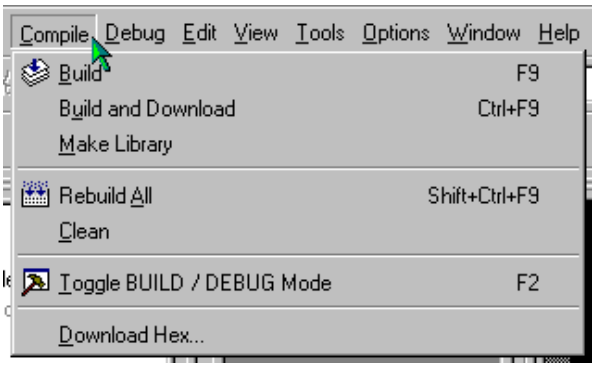
The "File" menu commands include the standard "New", "Open", "Save", "Save As", "Save All", and "Close" file commands. The print commands are also located here.

11.1.3 Module

A module is a single file that belongs to a project. Typically modules are subroutines. 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 old project to a new project by importing it or by using the "Cut" or "Copy" and "Paste" commands in the Module menus. You may set "Module Properties", "Create Modules", "Open Modules", "Save", "Close", or "Delete Modules" of the current project using the commands under the "Module" menu. The "Code Wizard" is not implemented yet.

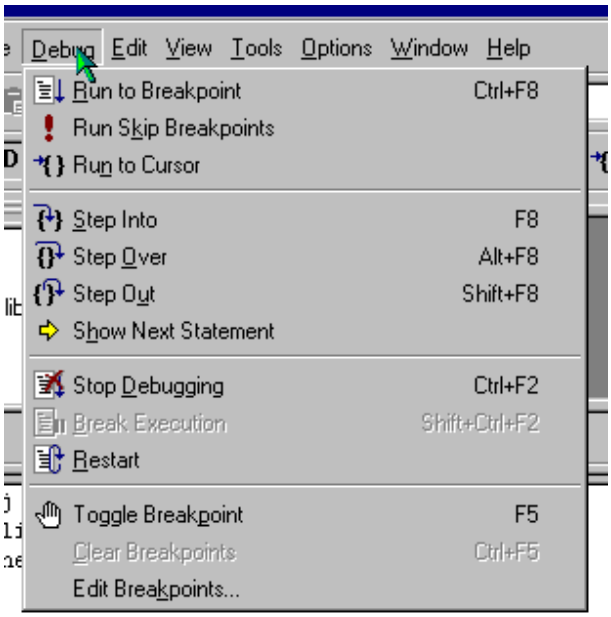
11.1.4 Compile

The "Compile" menu commands include "Build", "Build and Download", "Make Library", "Rebuild All", "Clean", "Toggle BUILD / DEBUG Mode", and "Download Hex". "Build" compiles the current project. If no project is open and the editor contains a file, this current file is compiled. "Build and Download" compiles the highlighted project and downloads it to the target board. "Toggle BUILD / DEBUG Mode" switches between the Build and Debug modes. "Rebuild All" and "Download Hex" are basic features that implement the stated command.



11.1.5 Debug

The “Debug” menu allows you to control the debug features of Reads51. You may “Edit Breakpoints”, “Toggle Breakpoints”, “Clear Breakpoints”, and “Run to”, “Run Skip”, “Step Into”, “Over” or “Out of Breakpoints”.

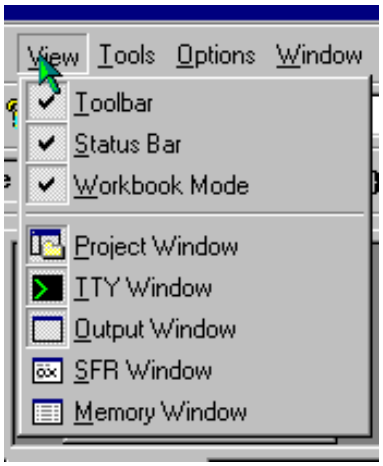


11.1.6 Edit

The “Edit” menu commands are the standard edit commands found in most programs. They allow you to “Redo”, “Cut”, “Copy”, “Paste”, “Find”, “Find Next”, “Replace”, and “Select All” the text.

11.1.7 View

The “View” menu commands are again the standard view commands with a couple of specific commands for Reads51 included. These commands allow you to open windows and customize the screen when working with Reads51.



11.1.8 Tools

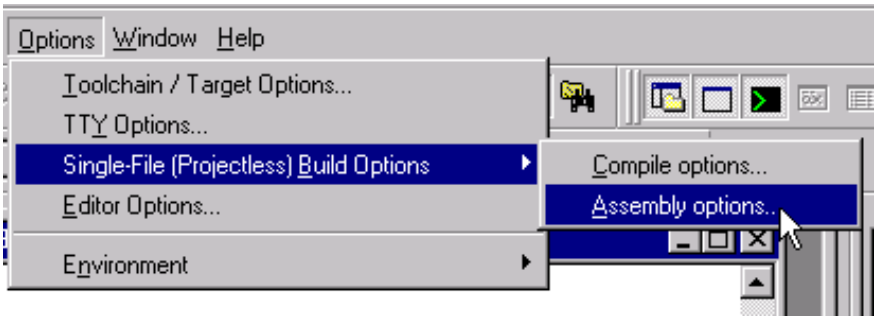
The “Tools” commands allow you to search in files for given strings with the “Find in Files” command, change the toolbars with the “Customize Toolbars” command, or “Burn RIC320 EEPROM” on one of our boards.

11.1.9 Options

The “Options” menu allows you to select the toolchain and target you want to use. It also allows you to select the“TTY Options”, the compile and assembly options for single files, “Editor Options”, and “Environment” options.

11.1.10 Window

These are the standard Window commands found in most programs; “Cascade”, “Tile”, “Arrange Icons”, and “Close All”.

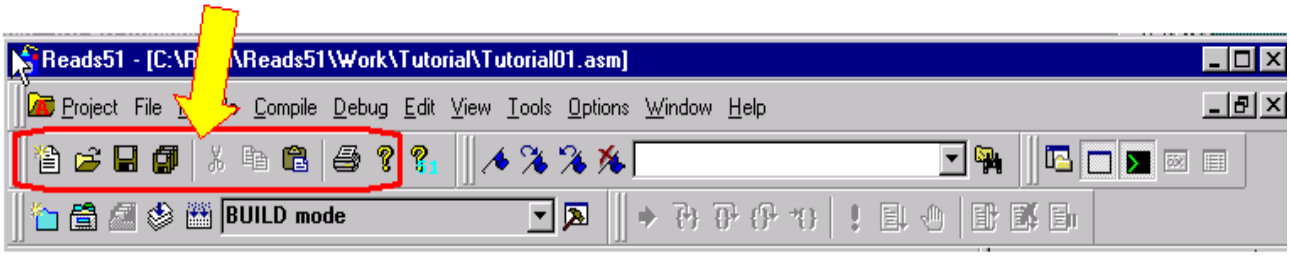


11.1.11 Help

A wide variety of information can be found in the Help files. We’ve added the MCS-51 instruction set, HTML help system, Quick Start, and updated our standard help files.

11.2 Toolbars

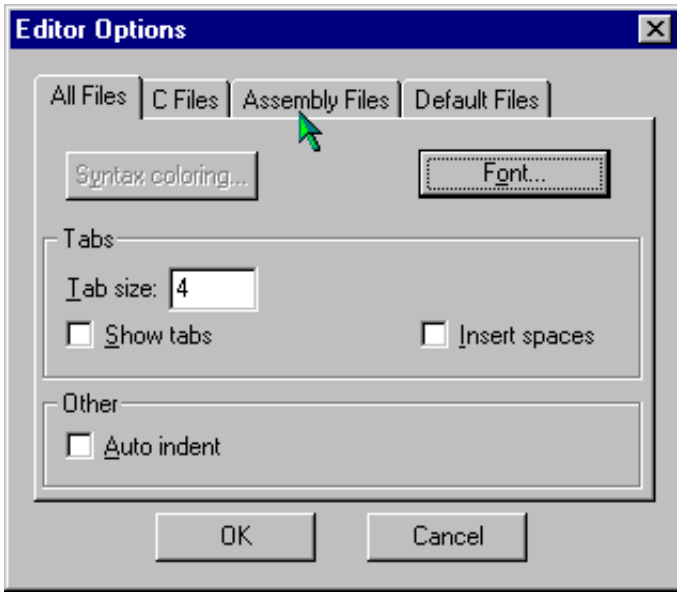
A Toolbar is a row of buttons at the top of the main window, which represent application commands. Clicking one of the buttons is a quick alternative to choosing a command from the menu. Many of the Toolbar buttons are the standard Windows buttons. “New”, “Open”, “Save”, “Save All”, “Cut”, “Copy”, “Paste”, “Print”, and “Help” are easily recognizable from other Windows programs.



Buttons on the toolbar activate and deactivate according to the state of the application, whether you are in the Build or Run/Debug Mode. Since toolbars are user customizable, it is better to investigate the function of the toolbar buttons for the current IDE by observing the ToolTips or the icons presented in the menus. For example, click on the "**Compile | Toggle BUILD/DEBUG Mode**" menu command. The corresponding toolbar button icon (a hammer) is shown next to the menu item. Clicking the menu item is equivalent to clicking on the corresponding toolbar button. Also, observe that F2 is given as the hot key to toggle the mode.

11.3 Editor

The editor uses a multiple document interface so that several files may be opened at a time. The editor window contains tabs in the bottom to quickly select the active child window. The tabs are especially useful if you maximize the active child window. You may use the Windows "drag-and-drop" feature to open any text file with the editor.



The editor uses standard Windows Notepad- or Windows Wordpad-style commands. In addition, the editor recognizes assembly and C syntax. Several editor settings as well as syntax highlighting may be customized by the "**Options | Editor Options**" menu.

The corresponding dialog lets you select fonts, set auto indenting, and specifying whether tabs should be replaced by a number of spaces. Note that the "All Files" tab in the dialog sets the properties globally, i.e. affects all other types of files. The check boxes under the "All Files" tab have three states. The checked and uncheck states override all other file type settings. In the grayed state, the properties are determined individually for each file type.

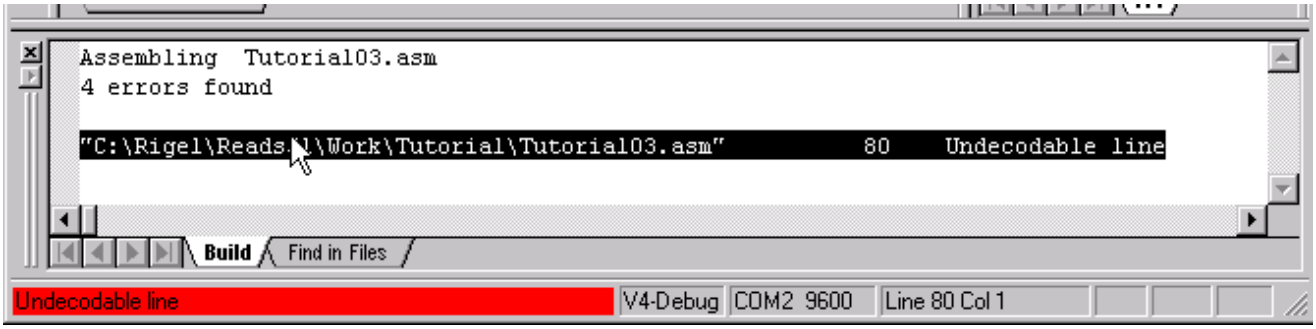
Syntax highlighting lets you specify the colors of keywords, strings, comments as well as default text and the background. The keywords to be highlighted are read from the files assembly.kwd, c.kwd, and default.kwd, found in the .\Bin directory. You may modify the set of keywords by opening these files in the editor and adding new keywords or removing existing ones.

11.4 TTY Window

The TTY Window is associated with a terminal emulation routine so that characters typed in the TTY window are sent to the serial port. Similarly, and the characters received from the serial port are displayed in the TTY window. The TTY Window properties are configurable using the "**Options | TTY Options**" menu. If the selected serial port is unavailable, the TTY Window displays the message "Disconnected."

11.5 Output Window

The output window has tabs to report the result of various activities. The "Build" tab shows the compiler or assembler results. Similarly, the "Find-in-Files" tab reports the results of searches from the Find-in-Files tool.



The results shown in the Output Window often relate to specific lines of source files. Simply double click on the output window results to open the source file and display the corresponding line. For example, if a build operation finds errors in the source, double clicking on the reported error takes you to the offending source line.

11.6 Tools

11.6.1 Find-in-Files

"Find-in-Files" is similar to the UNIX GREP (Get Regular ExPression) utility. It scans a specified set of files to find the occurrences of given strings. A drop-down list box and a button are placed on the default toolbar to facilitate "Find-in-Files". The results of the search are displayed in one of the tabs of the output window.

11.6.2 Run Preprocessor

The compiler and the assembler call the preprocessor automatically, as part of the build process. This menu command is provided mostly as a debugging aid or a teaching aid. The user may run code containing macros and compiler directives and observe the resultant file.

11.6.3 Customize Toolbar

The corresponding dialog allows you to define new toolbars, or add or remove buttons on existing toolbars. "Cool Look" refers to MS IE4-style dockable toolbars (rebar).

Under the dialog "Commands" tab, you may select any button and add it to an existing toolbar simply by dragging the button onto the toolbar. Similarly, you may remove buttons from an exiting toolbar by dragging the button away from the toolbar. Note that menu items may be added to any toolbar, just like any other button.

12 TUTORIALS

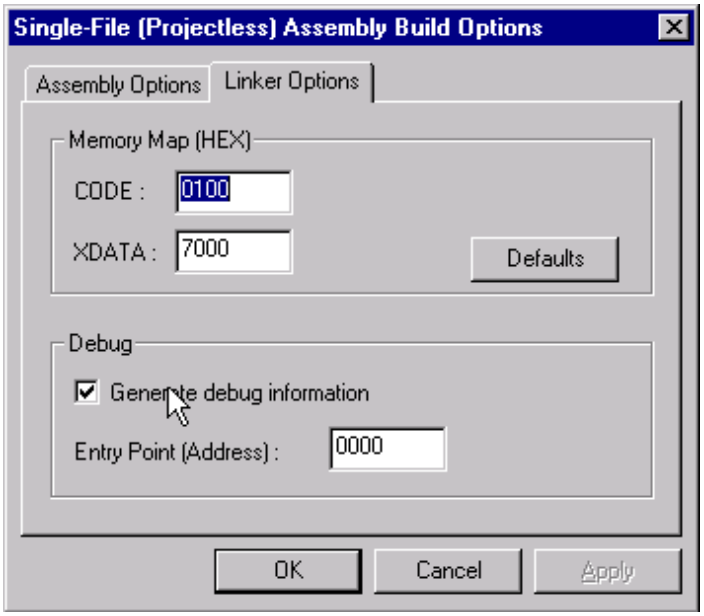
All of these tutorials can be found in the Reads\Work\Tutorial Directory. These are single files and will need to be opened using the “Files | Open File” command. Each tutorial builds on the concepts from the previous tutorial and therefore should be done in order.

12.1 Single Files

This is the first of six tutorials and is designed to show how to compile and debug single files. Most of the text below is found in the Tutorial0x.asm files.

Step 1: Open the tutorial file.

Go to the menu item “**File | Open File**” and open the file “Tutorial01.asm” in the Rigel\Reads51\Work\Tutorial directory. Click on the file and it will open in the editor window.



Step 2: Select Toolchain and Target.

Click on the menu item "**Options | Toolchain/Target Options**". Select "Reads51 V4 toolchain" and "RChipSim51".

Step 3: Specify Memory Map.

Click on the menu item "**Options | Single File Build Options | Assembly Options**". Select the tab "Linker Options". Specify the memory map to be CODE=0 and XDATA=0.

Step 4: Specify Debug Information to be Generated.

Again, using the menu item "**Options | Single File Build Options | Assembly Options**", Check the box "Generate debug information".

Step 5: Build (Assemble and Link).

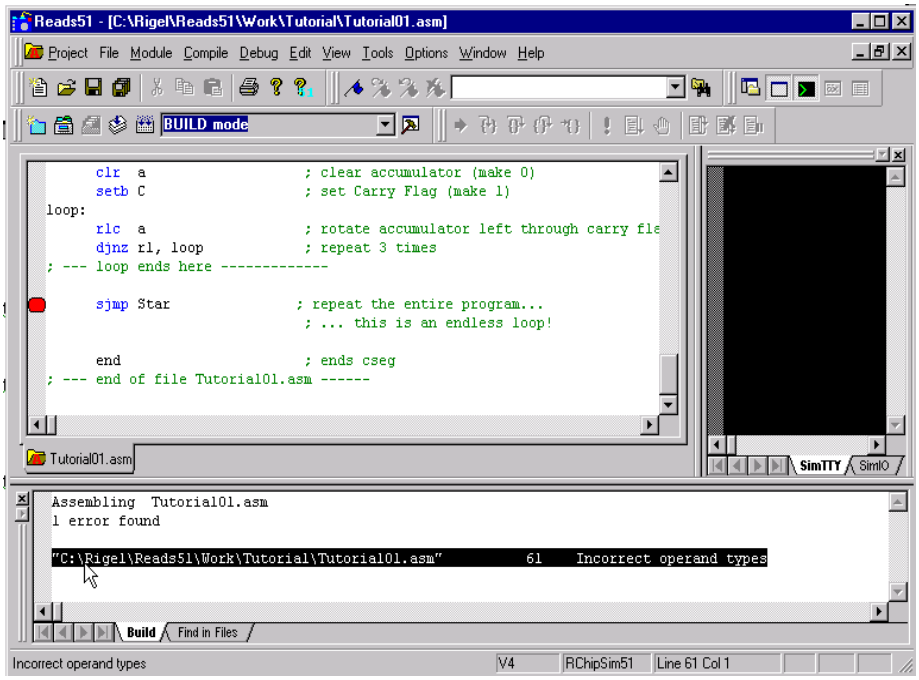
With “Tutorial01.asm” as the active window in the editor, click menu item "**Compile | Build**".

Step 6: Fix Errors.

The error "Incorrect operand types" is displayed in the output window. If the output window is not visible, click the menu item "**View | Output Window**". Double-click the error message. The instruction

```
sjmp Star
```

has an invalid label. Change "`Star`" to "`Start`" and rebuild the file (Repeat Step 5).



Step 7: Accelerator Keys and Toolbars.

It is cumbersome to use the menu for the various build and debug commands. As you get more familiar with Reads51 you may use the toolbars or the shortcut keys to invoke the various commands.

12.2 Debugging with RChipSim51

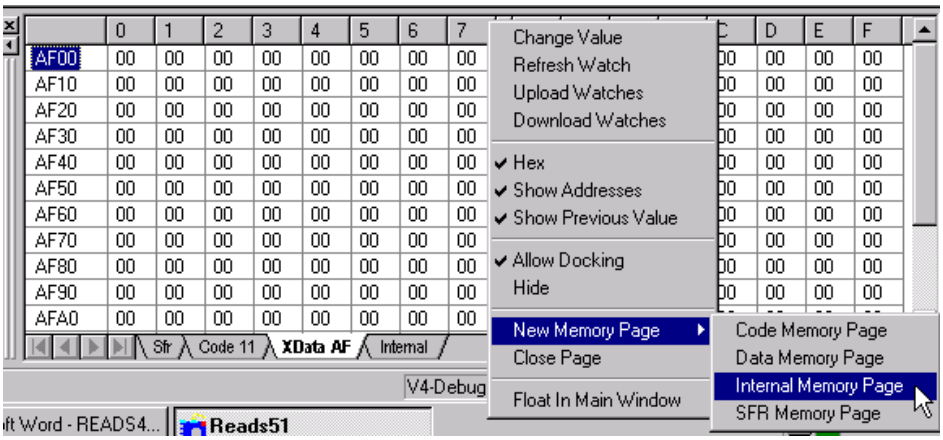
Step 1: Run "Tutorial02.asm"

Follow the steps 1-5 from Tutorial 5.1 and build the source file.

Step 2: Step Through the Program.
Click the menu item "**Compile | Toggle BUILD / DEBUG Mode**".
This loads the target (selected to be RChipSim in Step 1) with the generated HEX file.

Step 3: Open the SFR Window.
Click the menu item "**View | SFR Window**"
This enables the SFR Watch Window.

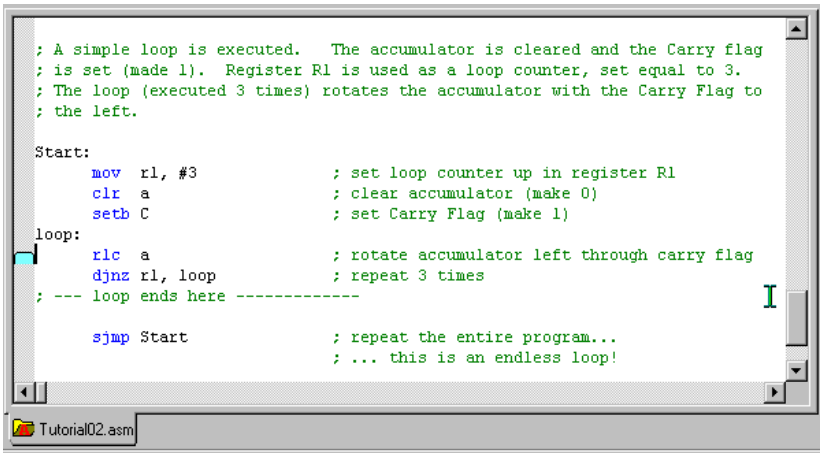
Step 4: Open the Memory Watch Window to View Internal Data Memory.
Click the menu item "**View | Memory Window**"
This enables the Memory Watch Window. Inside the Memory Window, right-click and select "New Memory Page". Specify "Internal Data Page".



Step 5: Single Step.
Click the menu item "**Debug | Step Into**".
The current instruction is indicated in the source window by an arrow. Similarly, the status bar (at the very bottom of the frame) shows the current instruction. When the instruction

```
mov  r1, #3
```

is executed and the arrow points to the following instruction (clr a) observe that the internal data memory page shows that location 1 (R1) holds the value 3. Continue to single-step (press F8) to execute more instructions. Note how R1 and ACC change.



Step 6: Set a Breakpoint.
Click on a valid instruction to move the caret (blinking vertical bar of the editor).
Click the menu item "**Debug | Toggle Breakpoint**".
A small blue mark appears next to the instruction.
Now click "**Debug | Run to Breakpoint**"
This executes all instructions up to the breakpoint. RChipSim51 supports an unlimited number of breakpoints. You may set other breakpoints and execute the program, stopping at each breakpoint.

Step 7: Clear All Breakpoints.
Move the caret to each breakpoint line and toggle the breakpoint.
You may remove all breakpoints with the menu item "**Debug | Clear Breakpoints**".

Step 8: Run to Cursor.

First click on a valid instruction to move the caret.
Then, click the menu item "**Debug | Run to Cursor**"

Step 9: Running and Stopping.

With no breakpoints set, click the menu item "**Debug | Run to Breakpoint**".
Since there are no breakpoints, RChipSim51 executes the instructions.
Click the menu item "**Debug | Break Execution**" to stop the execution.
You may now inspect the registers, single step, etc.

Step 10: Return to the Build Mode.

Click the menu item "**Compile | Toggle RUN/DEBUG Mode**".
Note that the source is not editable (is "read only") during debugging. Also note that the watch windows are closed and the output window is displayed in the build mode.

12.3 Debugging on a Rigel Board (RROS)

Step 1: Open the file "Tutorial03.asm".

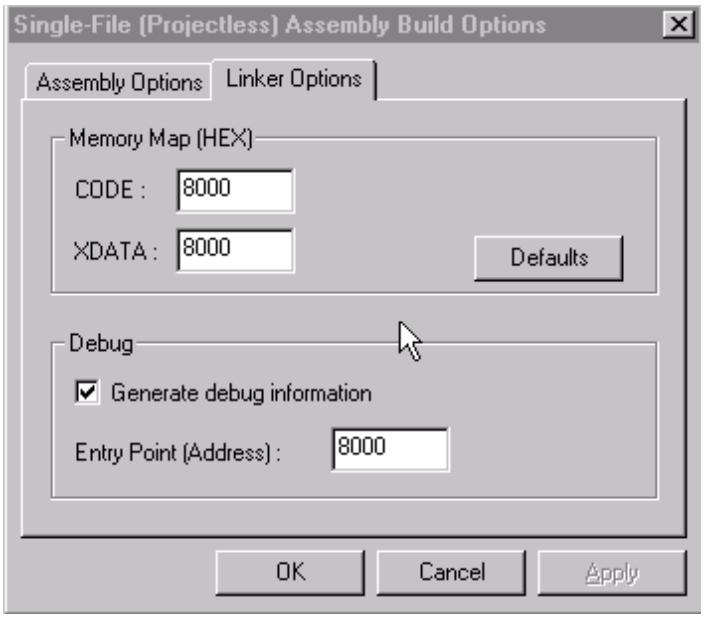
Step 2: Select Toolchain and Target.

Click on the menu item "**Options | Toolchain/Target Options**".
Select "Reads51 V4 toolchain" and "RROS".

Step 3: Specify Memory Map.

Click on the menu item "**Options | Single File Build Options | Assembly Options**".
Select the tab "Linker Options".
Specify the memory map to be CODE=8000 and XDATA=8000. Specify the entry point to be 8000. Note that the origin of the program is now 8000h, as specified by the line "cseg at 8000h", the first line of the program.

```
      cseg at 8000h     ; absolute
segment starting at (origin) 0
```



Click the menu item "**Compile | Clean**" before building the source.
It is a good idea to always remove any output files generated by a previous setting. The "Clean" command deletes these intermediate files. When rebuilt, the new memory map will take effect.

Step 4: Specify Debug Information to be Generated.

Again, using the menu item "**Options | Single File Build Options | Assembly Options**",
Check the box "Generate debug information".

Step 5: Open the TTY Window.

Click the menu item "**View | TTY Window**".
Press the Reset button on the board.
If you do not observe the monitor prompt, use the menu item "**Options | TTY Options**" to select an available port. Unless otherwise stated in the board's hardware manual, set the Baud rate to 9600.

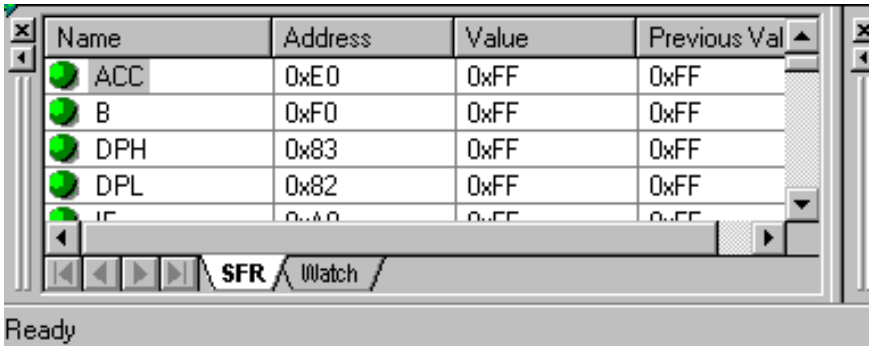
Step 6: Step Through the Program.

Click the menu item "**Compile | Toggle BUILD/DEBUG Mode**". This assembles the file and loads the target (selected to be RROS) with the generated HEX file. The Build and Debug modes have their own layouts. If the TTY window is not visible, open it as in the previous step.

Step 7: Open the SFR Watch Window.

Click the menu item "**View | SFR Window**"

This enables the SFR Watch Window.



Step 8: Open the Memory Watch Window to View Internal Data Memory.

Click the menu item "**View | Memory Window**"

This enables the Memory Watch Window. Inside the Memory Window. Right-click and select "New Memory Page". Specify "Internal Data Page".

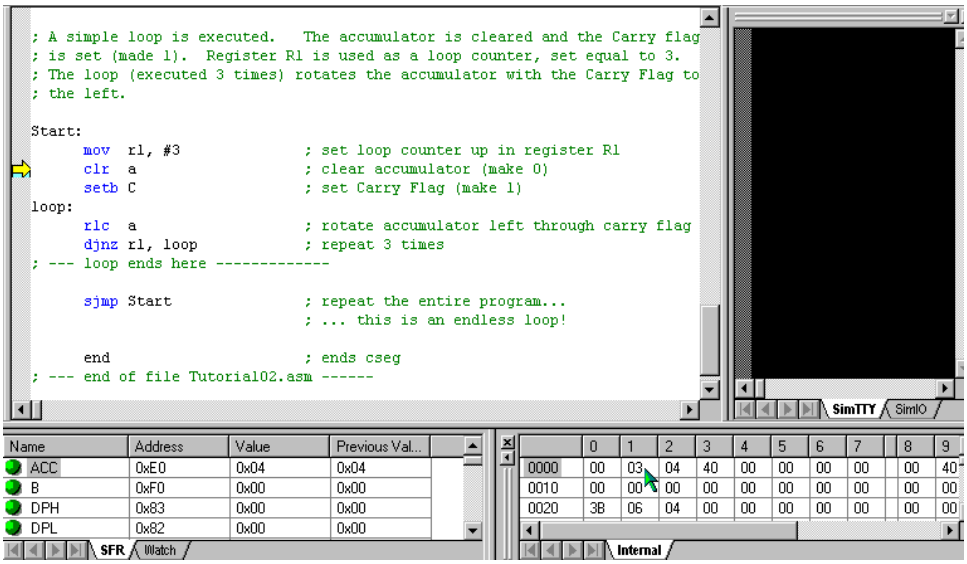
Step 9: Single Step.

Click the menu item "**Debug | Step Into**".

The current instruction is indicated in the source window by an arrow. Similarly, the status bar (at the very bottom of the frame) shows the current instruction. When the instruction

```
mov   r1, #3
```

is executed and the arrow points to the following instruction (clr a) observe that the internal data memory page shows that location 1 (R1) holds the value 3. Continue to single-step (press F8) to execute more instructions. Note how R1 and ACC change. Compared to RChipSim51, note that single stepping takes more time. After each step, the IDE communicates with the board to upload the watch values. This takes time.



Step 10: Run to Cursor.

RROS does not support multiple breakpoints. However, "Run to Cursor" allows you to execute code up to a given point. This command works as in Tutorial02.

Step 11: Running and Stopping.

The "Running" and "Stopping" commands are not available with the RROS target. When the board starts "Running" the program it stops inspecting the serial port and the IDE has no mechanism to stop the execution. If you run the program (without breakpoints) press the Reset button on the board to stop execution.

12.4 Watching Selected Variables During Debug

Step 1: Open "Tutorial04.asm".

This tutorial is an extension of Tutorial02.

Follow the steps in Tutorial02 to single step through the code.

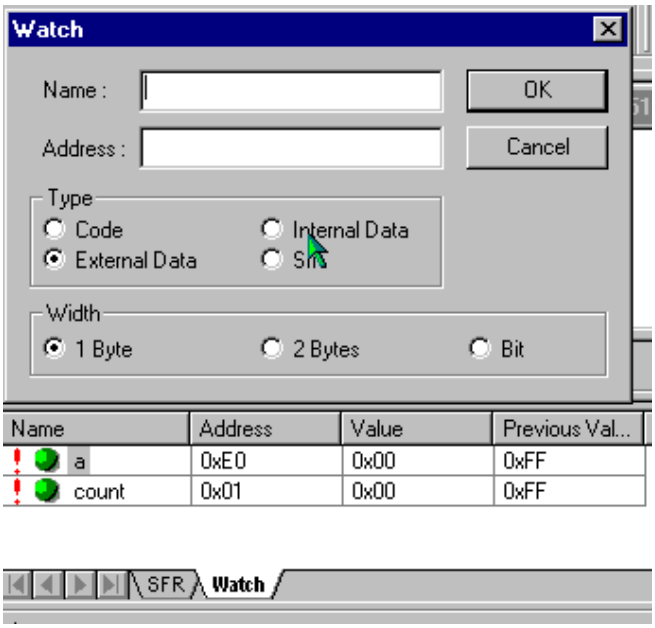
Step 2: Create a List of Selected Watch Variables

While in the debug mode, open the SFR window as in Tutorial02.
Click on the tab "Watch".
An empty window will appear.
Right-click to invoke the local menu. Select "Add Watch". Specify the following:

Name:	A
Address:	0xE0
Type:	SFR
Width:	1

This is the accumulator. Depending on your application, you may give more descriptive names to your variables.

You may find the addresses of the SFRs by clicking the SFR tab and observing their addresses.



Also note that you may hide the “Address Field” and the "Previous Value" field using the local menu. The values are updated as you step through the program. The values may be displayed in HEX or decimal, again determined by the local menu choice.

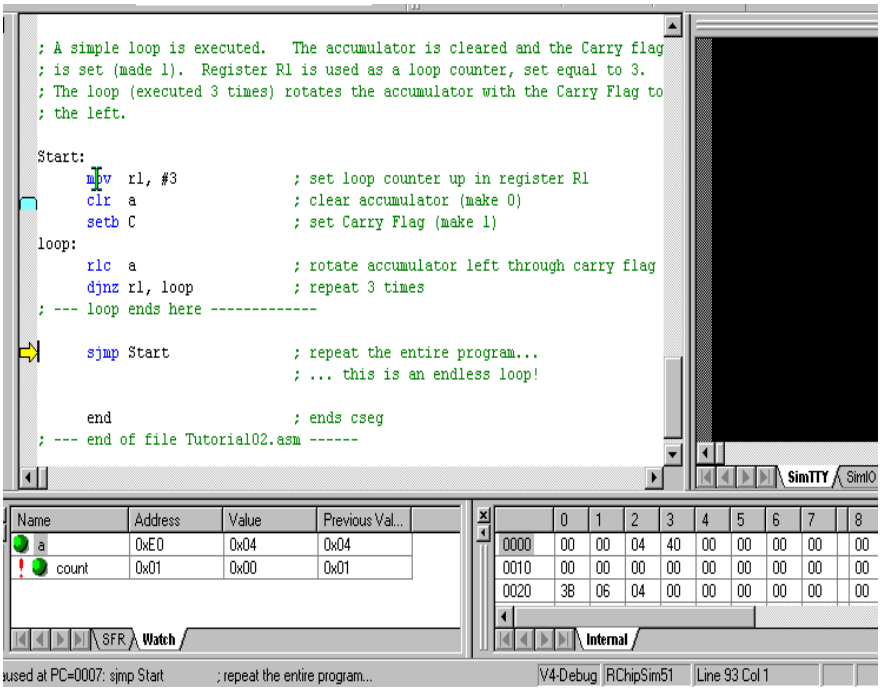
Step 3: Editing the Watch.

Double-click the name field or column ("A") of the added watch to edit its properties.
Double-click any other field (column) to change the value.
Add the following watch:

Name:	Count
Address:	1
Type:	Internal Data
Width:	1

This is register 1 (R1) which is decremented in the loop. The count is initialized to 3 and is decremented each time the accumulator is rotated.

At a breakpoint, select the watch "Count" by clicking on its name. Double-click its value field and change its current value. Continue single stepping to observe the effects.



Step 4: Editing the Memory Watch Window Entries.

Open the memory watch window as in Tutorial02 and open the "Internal Data Memory” page.
Again, at a break point, select the row you want to edit by clicking on the first column entry.
Double-click a cell to modify its value.

Step 5: Debugging on the Board.

Repeat the same steps using a Rigel board as in Tutorial03. Change the program origin to 8000h, and similarly use the "**Options | Single File Build Options | Assembly Options**" menu to change the memory map under the "Linker Options" tab.

12.5 Simulated I/O (SimIO)

Step 1: Open "Tutorial05.asm".
Select the RChipSim51 option.
Assemble the program "Tutorial05.asm" as in the previous tutorials.

Step 2: Open the simulated TTY and IO window (View | TTY Window).
Click the "SimIO" tab to see the ports buttons/indicators.

Step 3: Remove any breakpoints (Debug | Edit Breakpoints menu)
Run the program (Debug | Run to Breakpoint).
The program runs the endless loop. There are two inputs P1.0 and P1.1. The two outputs, P1.6 and P1.7 are computed from the inputs using the bitwise AND and OR operations.

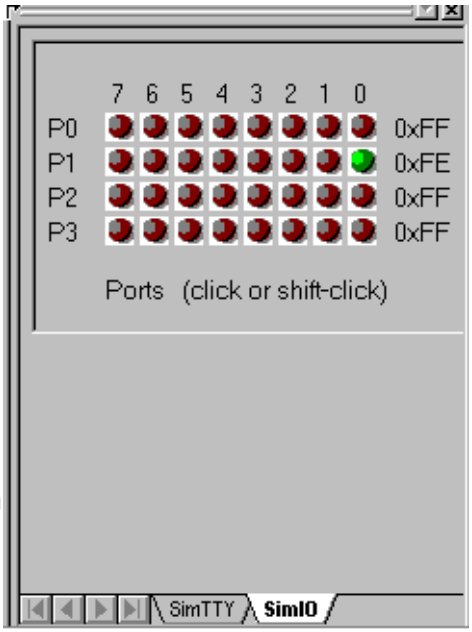
Step 4: Momentarily change an input.
Click and hold the mouse left button on P1.0.
Clicking on a button simulates grounding the corresponding port bit. Note that the ports have internal pull-up resistors. Their active state is 0 (grounded). P1.0 will remain green, indicating its active state, as long as you hold the Reset button down.
Also note that P1.6 becomes low, since P1.6 is the AND of P1.0=0 and P1.1=1. Also try clicking on P1.1.

Step 5: Toggle an input.
Hold the shift key and click on P1.0. P1.0 will remain active when you release the mouse button. Shift-clicks simulate toggle switches. If you now click P1.0 (without the shift button pressed), the bit will momentarily be 1. It will resume its active state as soon as you release the button.
Toggle P1.0 to remain active (low) and then click on P1.1 to make P1.7 active. (P1.7 is low when both P1.0 and P1.1 are low.)

Step 6: Terminate the program
Use the "**Debug | Stop Debugging**" or the "**Debug | Break Execution**" commands.

12.6 Simulated Serial I/O (SimTTY)

Step 1: Open "Tutorial06.asm".
Select the RChipSim51 option.
Assemble "Tutorial06.asm" as in the previous tutorials.



Step 2: Open the simulated TTY and IO window (View | TTY Window).
Click the "SimTTY" tab to see the ports buttons/indicators.

Step 3: Remove any breakpoints (Debug | Edit Breakpoints menu)
Run the program (Debug | Run to Breakpoint).
The program runs the endless loop. It waits for a character from the serial port. Once received, it echoes the character back.

Step 4: Type characters in the SimTTY window
Observe the response.

Step 5: Terminate the program
Use the "**Debug | Stop Debugging**" or the "**Debug | Break Execution**" commands.

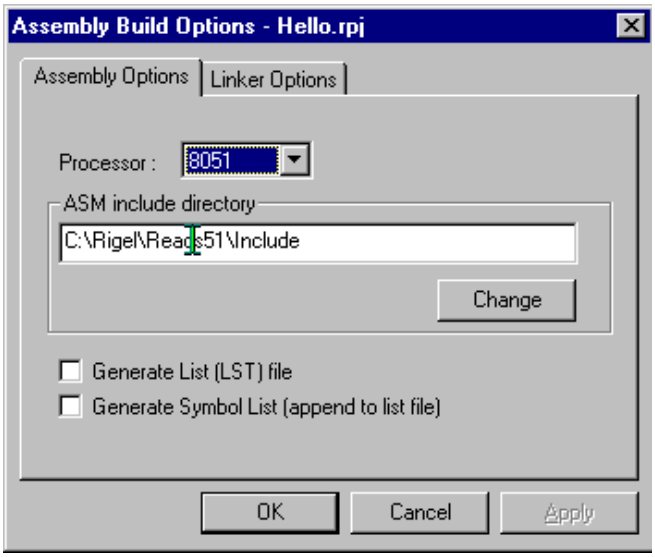
13 GENERATING HEX FILES

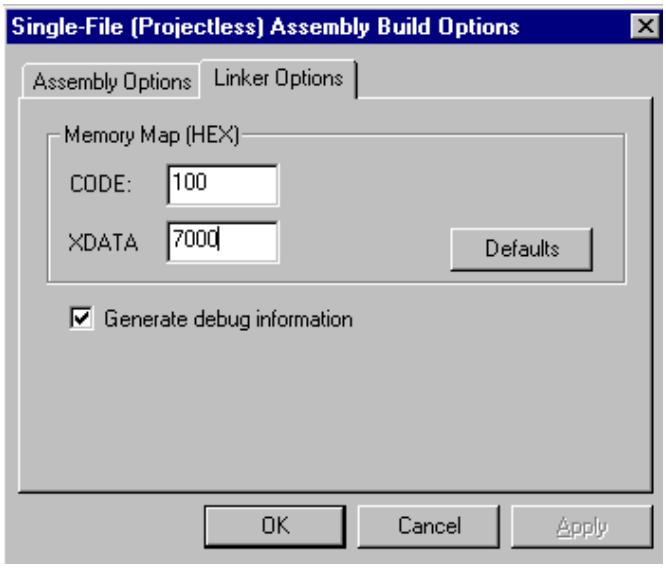
There are several ways to write and compile (or assemble) programs using the Reads51 environment. The alternatives depend on the target, the source, and whether the code is to be debugged.

1. Target: RROS, rChipSim51, or Embedded ROM.
2. Source: C source, relative assembly source, or absolute assembly source.
3. Debug: On or off.

The memory map of the generated HEX code depends on the linker options, found in the "Assembly Options" menu dialog box, under the "Linker Options" tag. If you are compiling a project, use the "**Project | Project Build Options | Assembly Options**" menu. If you are compiling a single file, use the "**Options | Single-File Build Options | Assembly Options**" menu.

The following parameters affect the way the final HEX code is generated:





1. The" Generate debug information" box must be checked if you want to debug the code in the Reads51 environment. Please note that currently Reads51 only supports low-level debugging. That is, single stepping, etc. are supported only at the assembly level.
2. The CODE origin. This parameter specifies the starting address of the code. Note that this parameter is meaningful only if relative assembly is used. Since the C compiler output is assembled using the relative assembler and the linker, this field must also be set correctly for C projects. C projects must be written to run from the lower 32K of memory. By default, the CODE parameter is set to 100 (hex) to allow room for the interrupt vectors. Note that the C compiler automatically places a jump instruction at the reset vector (address 0). Similarly, the C compiler places jump instructions at the specified addresses when interrupt routines are used.
3. The XDATA origin. This parameter specifies the starting address of the external data segment. The Reads51 C compiler requires external data memory. By default, XDATA is set to 7000 (hex)

The target you want to run the HEX code on will determine how the parameters will need to be set. If you intend to place the generated HEX code in ROM, use the same setting as those for the rChipSim51.

13.1 Running Code on a Rigel Board

Most Rigel boards in the RROS mode use 64K of overlapped CODE and XDATA memory. On the Rigel boards the RROS resides in the lower 32K memory and the 8051 interrupts are redirected to high memory (RAM) to the range FF00h...FFFFh. It is important to keep in mind that any HEX file downloaded through RROS is always placed into RAM. More specifically, RROS sets the most-significant-bit of the HEX record address. This way, HEX records with addresses in the range 8000h to FFFFh are downloaded in the usual manner to their respective addresses. On the other hand, HEX records with addresses in the range 0 to 7FFFh are downloaded to RAM with an added offset of 8000h.

The RUN/MON slide switch used on the Rigel 8051 boards determines the memory map. In the MON position, the first 32K of memory is mapped to the (RROS) EPROM and the second 32K of memory is mapped to RAM. In the RUN position, the blocks are swapped, with RAM in the lower 32K of memory, and the EPROM in the upper 32K of memory. Note that most boards have a red and a green LED. The red LED is turned on in the MON position while the green LED is turned on in the RUN position. The boards respond to the RROS monitor commands when the slide switch is in the MON position. With the slide switch in the RUN position the RROS is unavailable. For more information on the RROS please see the document "RROS Manual" available from the web at www.rigelcorp.com.

To run a program from low memory on a Rigel board press and hold the reset button while moving the slide switch from the MON to the RUN position. Verify that the green LED is turned on. Then release the reset button. The microcontroller responds to the reset by starting the execution at address 0.

In general, you may,

1. Compile your code with a start address in the first 32K block of memory and flip the slide switch to the RUN position, or,
2. Compile your code with a start address in the second 32K block of memory and run it through the RROS monitor.

C projects must be compiled with a start address in the first 32K block of memory. Assembly projects may have a start address in either the first or second block of 32K memory.

13.1.1 Running C Code

The default CODE and XDATA parameters are the best choice for compiling C projects to be executed in the Reads51 environment. After you compiler the code, switch to the Run / Debug Mode (Compile | Toggle BUILD / DEBUG Mode). The HEX code is automatically loaded to the board. You are now ready to run the program and observe its performance using the TTY window. To start execution on the board, move the slide switch to the RUN position, while keeping the reset button pressed. The microcontroller responds to the release of the reset button by starting the execution at address 0. Note that the slide switch in the RUN position also swaps the memory map, so that the RAM (containing the downloaded HEX code) occupies in the first 32K block, and the RROS EPROM occupies the second 32K block of the memory map. While the program is running in this fashion, the RROS monitor is unavailable, and thus, this mode does not support debugging.

13.1.2 Running Assembly Code with Start Address in the 0 to 7FFFh Range

Assembly projects with a start address in the first 32K of memory are downloaded to RAM by the RROS monitor and the Reads51 IDE when you toggle the IDE mode from BUILD to RUN/DEBUG. To start execution on the board, move the slide switch to the RUN position, while keeping the reset button pressed. The microcontroller responds to the release of the reset button by starting the execution at address 0. Note that the slide switch in the RUN position also swaps the memory map, so that the RAM (containing the downloaded HEX code) occupies in the first 32K block, and the RROS EPROM occupies the second 32K block of the memory map. While the program is running in this fashion, the RROS monitor is unavailable, and thus, this mode does not support debugging. On the other hand, all interrupt vectors are in RAM, which allows you to run interrupt routines without having to use the remapped vectors.

13.1.2.1 Running Relative Assembly Code (V4 Toolchain)

You may use the default CODE and XDATA parameters for compiling relative assembly projects and switch the memory map by toggling the MON/RUN slide switch as with running C code. Again, in this case, you must explicitly place a jump instruction at address 0 (the reset vector) to the entry point of your program. The following code is from the demo project RelativeAssembly02 in the \Work directory.

```
cseg at 0          ; cseg is the keyword to start an absolute code segment
ljmp    _main
end                ; each segment must terminate with an "end" directive
```

After you compiler the code, switch to the **Run / Debug Mode** (Compile | Toggle BUILD / DEBUG Mode). The HEX code is automatically loaded to the board through the RROS monitor. You are now ready to run the program and observe its performance using the TTY window. To start execution on the board, move the slide switch to the RUN position, while keeping the reset button pressed.

13.1.3 Running Assembly Code with Start Address in the 8000h to FFFFh Range

Assembly projects with a start address in the upper 32K of memory are downloaded to RAM by the RROS monitor and the Reads51 IDE when you toggle the IDE mode from BUILD to RUN/DEBUG. These programs may be run under the supervision of the RROS monitor, and thus, may be debugged. The CODE and parameter determines the entry point to the code under RROS supervision. That is, when the IDE enters the RUN/DEBUG mode, the program may be run or debugged by using the commands under the "Debug" menu. Check the "Generate debug information" box in the Linker tab under the "Assembly Options". Again, note that project build options are under the menu "**Project | Project Build Options**", whereas build options for individual files are under the "**Option | Single-File Build Options**". When running the programs under RROS supervision, the slide switch remains in the MON position.

The CODE and XDATA parameters must be in the upper 32K of memory. Note that since CODE and XDATA spaces overlap, use different values for the two parameters. Consider, for example, code of size a few kilobytes. Let CODE=8000 (hex) and XDATA=A000 (hex). This allows code to be up to 8KByte (A000h - 8000h = 2000h or 8K). The remaining 24K is then available for XDATA.

13.1.3.1 Running Relative Assembly Code (V4 Toolchain)

The linker groups all code segments and allocates them according to the selected parameters. If you use more than one code segment, or if you use libraries, the sequence of the various code segments are determined by the linker. In such cases, it is safer to place a jump instruction at the starting address (CODE parameters) to the entry point of your program. In simple cases with single code segments, this is not necessary. Assuming CODE=8000 (hex), the following code may be used:

```
cseg at 8000h     ; cseg is the keyword to start an absolute code segment
ljmp    _main     ; _main is the entry point to the program
end               ; each segment must terminate with an "end" directive
```

After you compiler the code, switch to the **Run / Debug Mode** (Compile | Toggle BUILD / DEBUG Mode). The HEX code is automatically loaded to the board through the RROS monitor. If the program is compiled with the debug option, you may run the program using the commands under the "Debug" menu.

13.2 Running Code with rChipSim51

rChipSim51 simulates an 8051 which starts execution after reset. That is, execution always starts at address 0. Moreover, rChipSim51 assumes that the code and external data spaces are separate (non-overlapping). In this respect, generating code to be executed with rChipSim51 is almost always the same as generating code to be placed in ROM. Care must be taken only if code and external data memory spaces overlap in the hardware implementation. Select the rChipSim51 target in the "**Options | Toolchain | Target Options**".

13.2.1 Running C Code

The default CODE and XDATA parameters are the best choice for compiling C projects to be executed in the Reads51 environment. After you compiler the code, switch to the **Run / Debug Mode** (Compile | Toggle BUILD / DEBUG Mode). The HEX code is automatically loaded to the chip simulator. You are now ready to run the program and observe its performance using the SimTTY and SimIO windows. Although you may debug (e,g, single step) generated assembly code, debugging C projects is not recommended.

13.2.2 Running Relative Assembly Code (V4 Toolchain)

The default CODE and XDATA parameters are also the best choice for compiling relative assembly projects to be executed in the Reads51 environment. Note that you must explicitly place a jump instruction at the reset vector to direct the program execution to the entry point of your application. The following code is from the demo project RelativeAssembly02 in the \Work directory.

```
cseg at 0         ; cseg is the keyword to start an absolute code segment
ljmp    _main     ; _main is the application's entry point
end               ; each segment must terminate with an "end" directive
```

After you compiler the code, switch to the **Run / Debug Mode** (Compile | Toggle BUILD / DEBUG Mode). The HEX code is automatically loaded to the chip simulator. You are now ready to run the program and observe its performance using the SimTTY and SimIO windows. You may also debug (e.g. single step) the code and watch memory and SFRs.

14 READS51 v4 TOOLCHAIN

The Reads51v4 toolchain contains Rigel's relative assembler that was introduced with Reads51 v4.00 (1999). The IDE views the toolchains selection as is a global choice that affects all build operations of all open projects or single-file sources. Use the "Options | Toolchain Options" menu to specify the toolchain.

The relative assembler generates Intel HEX records from assembly source files in two steps. First the assembler generates object files in the Intel OMF-51 format. The object files are said to be relative, or relocatable, since they are not specified to be placed in any constant memory location in code memory. All memory-specific information is left out. The decision about where in the memory map the code is to be placed is made later. In this sense, the object files may be viewed as modules which may be placed anywhere in the memory map of the 8051. Accordingly, the object files contain the so-called "fixup" records. These records specify how references to memory locations need to be modified once the final location of the code is determined.

The second step in generating HEX code is the link step. Here, the memory locations are determined. The linker combines the object files and performs the fixup operations. The CODE and XDATA start addresses are perhaps the most important two parameters the linker needs.

14.1 Preprocessor

The macro preprocessor may be used with any type of file, assembly, C, or any other programming language. Its syntax is C-like. The macro preprocessor supports definitions, macros, and conditional compilation. The preprocessor directives start with the pound sign ('#'). The preprocessor may be viewed as the first step in preparing the source for compilation or assembly. The include files are inserted, the macros are substituted, and the conditional compilation directives are used to further include or exclude blocks of the source. The output of the preprocessor is a single source file.

#include "file name"
Inserts the specified file. The file should be in the current directory or on the path.#include <file name>

#include <file name>
Inserts the specified file. The file should be in the current directory, on the path, or in the designated default include directory. The default include directories are specified by the project options of the current project.

#define
Equates a symbol to another.
For example,

```
#define MAX 10
```

defines MAX to be equivalent to 10. The preprocessor replaces all instances of MAX with 10. The define directive may include arguments. These are called macro definitions. For example, in the following code,

```
#define ADD(a,b) (a+b)
.
.
X=ADD(acc, 20)
.
.
```

ADD(acc, 20) is replaced by

```
X=(acc+20)
```

#undefine
Removes a previously defined symbol from the list.

#ifdef
Includes the following block of source if the specified symbol is previously defined. Consider, for example,

```
#define DEBUG
.
```

```
.
#ifdef DEBUG
mov a, TMOD
#else
clr a
#endif
.
.
```

The output of the preprocessor will include the line

```
mov a, TMOD
```

but not the line

```
clr a
```

because DEBUG was previously defined.

#ifndef
Includes the following block of source if the specified symbol is not previously defined.

#else
This directive is used with the #ifdef directive. The following block of source is included if the specified symbol is not previously defined.

#endif
Delimits the #ifdef directive or #ifdef/#else pair of directives.

14.2 C Compiler

The C compiler is a SmallC-compatible compiler that generates MCS-51 relative assembly language from C source. The output is intended to be assembled by the Reads51v4 relative assembler and subsequently be linked by the Reads51v4 linker.

The C compiler has some of the limitations of SmallC. However, it also introduces some significant extensions and improvements over standard SmallC.
The C-Compiler's limitations (SmallC has these same limitations)

1. Structures and Unions are not implemented
2. Only one-dimensional arrays are allowed.
3. Only one level of indirection (pointer) is allowed.
4. Only int and char types are allowed.

Reads51v4 C-Compiler improvements

1. Uses the more modern (ANCI C) function argument definition syntax.
2. Arguments are passed to the functions in the C convention. This allows a variable number of arguments to be passed on to functions, such as in printf().
3. Supports MCS-51 interrupts.
4. Supports function prototypes.
5. Supports the void type.
6. Uses Rigel's proprietary macro preprocessor.
7. Supports sfr and sfrbit types.

The Appendix titled "A Brief Review of C" gives an overview of the language. Please refer to the various demos in the work directory for further examples.

14.3 Relative Assembler (Reads51v4 Toolchain)

The relative assembler generates Intel HEX records from assembly source files in two steps. First the assembler generates object files in the Intel OMF-51 format. The object files are said to be relative, or relocatable, since they are not specified to be placed in any constant memory location in code memory. All memory-specific information is left out. The decision about where in the memory map the code is to be placed is made later. In this sense, the object files may be viewed as modules which may be placed anywhere in the memory map of the

8051. Accordingly, the object files contain the so-called “fixup” records. These records specify how references to memory locations need to be modified once the final location of the code is determined.

14.3.1 Constants

Decimal constants are written as regular numbers.
Hexadecimal constants include numbers 0 to 9 and the letters a to f. They must start with a number and be terminated by the letter h (or H). Constants are case insensitive, e.g. 0ah is the same as 0aH, 0Ah, or 0AH. Hexadecimal numbers may also be written in ‘C’ language syntax with a preceding “0x” and no terminating ‘h’. For example

```
0x100
```

is `100h` or `256` in decimal.

Binary constants may include only the numbers 0 and 1. They must be terminated by the letter ‘b’ (or ‘B’).

```
101b or 101B
```

ASCII constants are written within single quotes, such as

```
'A'.
```

String constants are written within double quotation marks.

```
db "A line feed (ASCII 10) and a null (zero) follow this string.", 10, 0
```

14.3.2 Expressions

Basic arithmetic and logic operations are supported in a C-like syntax. 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.

Binary arithmetic operations: `*, /, %, +, -, <<, >>`

```
mov a, #(1+2)                   ; addition
mov a, #(1-2)                   ; subtraction
mov a, #(2*2)                   ; multiplication
mov a, #(8/2)                   ; division
mov a, #(1%2)                   ; modulus (remainder)
mov a, #((1+2)*(8-2))           ; use parentheses
mov a, #(1<<2)                  ; shift left
mov a, #(0x100>>4)              ; shift right
```

Unary arithmetic operations: `-`

```
mov a, #-1                      ; unary minus
```

Binary bitwise (Boolean) operations: `&, |, ^`

```
mov a, #(1&2)                   ; bitwise and
mov a, #(1|2)                   ; bitwise or
mov a, #(1^2)                   ; bitwise exclusive or (exor)
```

Unary bitwise (Boolean) operations: `~`

```
mov a, #(~1)                    ; one's complement
```

Binary logic (Boolean) operations: `&&, ||`

```
mov a, #(1&&2)                  ; logic and
mov a, #(1||2)                  ; logic or
```

Conditions

```
mov a, #(1==2)                  ; equal
mov a, #(1!=2)                  ; not equal
mov a, #(1<2)                   ; less than
mov a, #(1<=2)                  ; less than or equal
mov a, #(1>2)                   ; greater than
```

```
mov a, #(1>=2)                ; greater than or equal
```

Unary logic (Boolean) operations: ! |

```
mov a, #(!1)                  ; logical not
mov a, #low(0x101*0x3)

here:
   mov a, #high($)
   mov a, #((here&0xFF00)>>8)
   mov a, #(($&0xFF00)>>8)
   mov a, #($&0xFF00)
```

14.3.3 Functions

The functions low() and high() of Reads51v3.x are preserved for backward compatibility. Note that these may also written as expressions. Low(N) is the same as (N & 0xFF) and High(N) is the same as (N>>8).

low()

Function: extracts the low byte of a word constant.
Description: Given the word (2-byte value) N, the value of low(N) is equal to the low byte of N.
Example:

```
LABEL:
   .
   .
   MOV  A, LOW(LABEL)         ; LABEL is a 16-bit address
   .                          ; get the low byte of this address
```

high()

Function: extracts the high byte of a word constant.
Description: Given the word (2-byte value) N, the value of high(N) is equal to the high byte of N.
Example:

```
LABEL:
   .
   .
   MOV  A, HIGH(LABEL)        ; LABEL is a 16-bit address
   .                          ; get the high byte of this address
```

14.3.4 Pseudo Operations

The relative assembler of Reads51v4.x uses a preprocessor to support include files, macro definitions and conditional assembly. Most pseudo operations are related to how code segments and modules are defined in the MCS-51 assembly language. Pseudo operation (pseudo ops) are used in assembly language, similar to machine language instructions. Unlike machine language instructions, pseudo operations do not correspond to a given processor operation. Rather, pseudo ops are directives to the assembler. Most of the pseudo ops are related to segment and module definitions. Also, note that some pseudo ops are used in more than one context.

14.3.5 Constant Definitions

EQU (pseudo op)

Function: constant definition
Description: Assigns symbols to constants. EQU pseudo ops improve the readability of your code by using more meaningful variable names rather than numerical addresses. It also allows you to quickly reassign the variables by simply modifying the EQU definition rather than making changes for all occurrences of the variable.
Example:

```
COUNT  EQU  28h              ; internal register 28h is called "COUNT"

  .
  .
MOV   COUNT, TL0             ; save count
  .
  .
```

```
MOV   A, COUNT                ; get "count"
  .
  .
```

14.3.6 Initialized Data Storage

DB (pseudo op)

Function: data storage

Description: The data bytes or strings of ASCI characters are placed starting from the current memory location. Strings must be delimited with double quotations. Strings may not include the comma (',') character. Strings and constants may be combined, separated by commas. Data defined by each DB pseudo op must be 255 bytes or less. For larger data blocks, use two or more DB pseudo ops.

Note that only data defined in a code segment may be initialized. Internal and external data is, by the nature of the 8051 architecture, volatile, and thus does not retain initialization values.

Example:

```
DB  0,1,2,3,4                 ; defines 5 bytes (ov value 0 to 4)
DB  "hello"                   ; defines 5 bytes of value'h','e','l','l','o'

DB  "dog", 0                  ; defines 4 bytes of value 'd', 'o', 'g', 0

; since commas are not allowed within strings,
; the following uses the ASCI value 2Ch instead.

DB  "one", 02Ch, "two"        ; the string "one,two"DW
```

DW (pseudo op)

Function: data storage

Description: The data words or strings of ASCI characters are placed starting from the current memory location. Each data word occupies two bytes. Each character of the string is kept in two bytes, the ASCII value of the character in the low byte, and 0 (zero) in the high byte. Strings must be delimited with double quotations. Strings may not include the comma (',') character. Strings and constants may be combined, separated by commas. The DB pseudo op is usually more suitable for defining strings. Data defined by each DW pseudo op must be 255 bytes or less. For larger data blocks, use two or more DW pseudo ops.

Note that only data defined in a code segment may be initialized. Internal and external data is, by the nature of the 8051 architecture, volatile, and thus does not retain initialization values.

Example:

```
DW  1234h,0ABCDh              ; defines 2 words (4 bytes)
DW  "dog", 0                  ; defines 4 words (8 bytes)
```

DS (pseudo op)

Function: data storage

Description: DS reserves a block of data. The data block may optionally be initialized to a given value.

Note that only data defined in a code segment may be initialized. Internal and external data is, by the nature of the 8051 architecture, volatile, and thus does not retain initialization values.

Example:

```
DS  10                        ; reserves 10 bytes
DS  10 << 0xFF                ; reserves 10 bytes and initializes all to 0xFF
```

DBIT (pseudo op)

Function: bitwise data storage

Description: Reserves a block of bits in internal bit addressable memory. The block of bits may be referenced by an optional label.

Example:

```
USER_FLAG:
DBIT  1                         ; defines 1 bit at location "USER_FLAG"
IO_COPY:
DB  0x18                        ; reserves a block of 24 bits
```

14.3.7 Code Origin and Offset

AT (pseudo op)

Function: sets absolute segment origin

Description: Absolute segment origins are determined at the source level. The AT pseudo op is used in conjunction with one of the absolute segment definition directives CSEG, XSEG, DSEG, ISEG, or BSEG. Segments must be terminated by an END directive.

Example:

```
CSEG AT 0x2000                  ; starts an absolute code segment at address
                                ; 2000h
  .
  .
  .
  .
END
```

ORG (pseudo op)

Function: sets segment origin or offset

Description: The effect of the ORG directive depends on the type of the current segment. If the current segment is an absolute segment, then ORG specifies an origin. That is, the address of the instruction that follows. If the current segment is a relative segment, then ORG specifies an offset from the beginning of the segment.

Note that the absolute address of a relative segment is not determined until the end of the linking process.

Example:

```
CSEG                            ; absolute segment
ORG 0x2000                      ; origin at 2000h


has the same effect as
CSEG AT 0x2000END
```

END (pseudo op)

Function: terminates an absolute or relative segment.

Description: In most cases the assembler is smart enough to end the current segment whenever a new segment is initiated. In order to avoid ambiguities, it is safer to always terminate a segment by an END directive.

Example:

```
SERIAL SEGMENT CODE
RSEG SERIAL                     ; start relative segment "SERIAL"
.
.
.
.
.
END                             ; end of current relative segment
CSEG AT 0x2000                  ; starts an absolute code segment at address
2000h
.
.
.
.
```

```
      .
      END                           ; end the absolute segment
      RSEG SERIAL                   ; re-open segment "SERIAL"
      .
      .
      .
      .
      .
      END                           ; end of relative segment
```

14.3.8 Absolute Segments

CSEG (pseudo op)

Function: defines an absolute code segment.

Description: CSEG defines and starts a new absolute code segment. Optionally, the absolute address of the segment origin may be specified using the AT directive.

Example: The following segment contains a simple subroutine that inspects the value in the accumulator and returns 0 if the accumulator value is even, and 0FFh if odd. The segment is placed in code memory at address 2800h.

```
      CSEG AT 0x2800                ; start a new code segment

      Odd:
            jb acc.0, IsOdd
            clr  a
            ret
      IsOdd:
            mov  a, #0xFF
            ret
      END                           ; end the segment
```

XSEG (pseudo op)

Function: defines an absolute external data segment.

Description: XSEG defines and starts a new absolute data segment. Optionally, the absolute address of the segment origin may be specified using the AT directive. Note that only the code segment may contain initialized data. The segment defined by an XSEG directive may reserve data bytes or words to be written to or read from during program execution.

Example: The following segment contains a simple subroutine that defines data in external memory. A separate code segment contains code to modify the data.

```
          XSEG AT 0x8000            ; start a new external data segment

      X:  DS 1                      ; byte variable X
      Y:  DS 2                      ; word variable Y
      A:  DS 10                     ; array A contains 10 bytes

            END                     ; end the segment

          CSEG AT 0x2000            ; start a new code segment

          mov  dptr, #X             ; byte variable X
          movx a, @dptr             ; read X
          inc  a                    ; increment X
    movx @dptr, a                   ; write incremented X back to external    ; data

            END                     ; end the segment
```

DSEG (pseudo op)

Function: defines an absolute internal direct data segment.

Description: DSEG defines and starts a new absolute direct data segment. Note that directly addressable internal memory of the 8051 architecture includes the 128 internal data memory and the special function registers. Portions of the internal data space are also addressable as the register banks or bit addressable.

Optionally, the absolute address of the segment origin may be specified using the AT directive. Note that only the code segment may contain initialized data. The segment defined by a DSEG directive may reserve data bytes or words to be written to or read from during program execution.

Example: The following segment contains a simple subroutine that defines data in direct memory. A separate code segment contains code to modify the data.

```
        DSEG AT 0x70              ; start a new external data segment

    X:  DS 1                      ; byte variable X
    Y:  DS 2                      ; word variable Y
    A:  DS 10                     ; array A contains 10 bytes

        END                       ; end the segment

        CSEG AT 0x2000            ; start a new code segment

        mov  a, X                 ; read byte variable X
          mov  b, #3
          mul  ab                 ; X*3
          mov  X, a               ; write X*3 back to internal data memory

    END                           ; end the segment
```

ISEG (pseudo op)

Function: defines an absolute internal indirect data segment.

Description: ISEG defines and starts a new absolute indirect data segment. Note that indirectly addressable internal memory of the 8051 architecture is the upper 128 internal data memory.

Optionally, the absolute address of the segment origin may be specified using the AT directive. Note that only the code segment may contain initialized data. The segment defined by a DSEG directive may reserve data bytes or words to be written to or read from during program execution.

Example: The following segment contains a simple subroutine that defines data in internal indirect memory. A separate code segment contains code to modify the data.

```
  ISEG AT 0xF0                    ; start a new internal indirect data segment

    X:  DS 1                      ; byte variable X
    Y:  DS 2                      ; word variable Y
    A:  DS 10                     ; array A contains 10 bytes

        END                       ; end the segment

        CSEG AT 0x2000            ; start a new code segment

        mov  r0, #X               ; address of X
        mov  a, @r0               ; read byte variable X
        mov  b, #3
        mul  ab                   ; X*3
        mov  @r0, a               ; write X*3 back to internal data memory

        END                       ; end the segment
```

BSEG (pseudo op)
Function: defines an absolute bit segment.
Description: BSEG defines and starts a new absolute bit segment. Note that bit addressable memory of the 8051 architecture is located in internal data memory, bytes 20h to 2Fh.

Optionally, the absolute address of the segment origin may be specified using the AT directive. Note that only the code segment may contain initialized data. The segment defined by a BSEG directive may reserve data bits to be written to or read from during program execution.

Example: The following segment defines bits. The segment is placed in code memory at address 2800h to read from and write to the defined bits.

```
      BSEG AT 0                    ; start a new external data segment

    X:      DBIT 1                 ; bit variable X
    FLAGS:  DBIT 8                 ; array FLAGS contains 8 bits

      END                          ; end the segment

      CSEG AT 0x2000               ; start a new code segment

      clr  X                       ; clear bit X
           mov  C, X               ; read X into the carry flag
           anl  C, (FLAGS+3)       ; logic and of X and FLAGS bit 3
           mov  (FLAGS(2), C       ; write result to FLAGS bit 2
      END                          ; end the segment
```

14.3.9 Relative Segments
CODE (keyword)
Function: refers to a relative code segment.
Description: CODE identifies the segment as a relative code segment. Code segments are placed in code memory at link time. The CODE keyword is used in declaring code segments, as below.

```
      Main      segment code
```

Also, the CODE keyword is used in identifying the type of external references. For example,

```
      extern    code    init_8031   ; function (label)
```

Example: Refer to the demo project RelativeAssembly01 in the work directory for an example.

XDATA (keyword)
Function: refers to a relative external data segment.
Description: XDATA identifies the segment as a relative external data segment. External data segments are placed in external data memory at link time. The XDATA keyword is used in declaring code segments, as below.

```
      Prompt    segment xdata
```

Also, the XDATA keyword is used in identifying the type of external references. For example,

```
      extern    xdata   sz          ; external (RAM) data (symbol)
```

Example: Refer to the demo project RelativeAssembly01 in the work directory for an example.

DATA (keyword)
Function: refers to a relative internal direct data segment.
Description: DATA identifies the segment as a relative internal direct data segment. Internal direct data segments are placed in internal memory at link time. The DATA keyword is used in declaring internal direct data segments, as below.

```
      PWM       segment data
```

Also, the DATA keyword is used in identifying the type of external references. For example,

```
extern    data    PWM         ; PWM value is saved in internal RAM
```

Example: Refer to the demo project RelativeAssembly01 in the work directory for an example.

IDATA (keyword)
Function: refers to a relative internal indirect data segment.
Description: IDATA identifies the segment as a relative internal indirect data segment. Internal indirect data segments are placed in internal memory at link time. The IDATA keyword is used in declaring internal indirect data segments, as below.

```
PWM       segment idata
```

Also, the IDATA keyword is used in identifying the type of external references. For example,

```
extern    idata    PWM        ; PWM value is saved in internal RAM
```

Example: Refer to the demo project RelativeAssembly01 in the work directory for an example.

BIT (keyword)
Function: refers to a relative bit segment.
Description: BIT identifies the segment as a relative bit segment. Bit segments are placed in bit addressable internal memory at link time. The BIT keyword is used in declaring bit segments, as below.

```
FLAG_1    segment bit
```

Also, the BIT keyword is used in identifying the type of external references. For example,

```
extern    bit    FLAG_1
```

Example: Refer to the demo project RelativeAssembly01 in the work directory for an example.

14.3.10 Modules and Intermodule Linkage

RSEG (pseudo op)
Function: starts a relative segment.
Description: Relative segments must first be declared. Then, the RSEG directive instructs the assembler to start placing the following instructions in the corresponding segment.

Example: Refer to the demo project RelativeAssembly01 in the work directory for an example.

```
MainCode    segment code

      cseg at 0              ; start an absolute code segment
      ljmp    main           ; branch to main upon reset
      end                    ; end segment

      rseg    MainCode       ; rseg is the keyword to start a relative
                             ; segment
main:                        ; entry upon reset
.
.
.
.
.

end                          ; terminate segment MainCode
```

14.3.11 EXTERN IMPORT (pseudo op)

Function: identify labels or symbols which are defined in another module.
Description: EXTERN is used with an identifier of the relative segment type to specify that the given labels or symbols are defined in other modules. IMPORT is an alternative keyword that is interpreted as EXTERN. EXTERN (IMPORT) and PUBLIC (EXPORT) pairs provide the primary mechanism for multimodule programming. They allow symbols (variables) or labels (code addresses) to be publicized (exported) by one module and imported (by extern) by another. For example, a

function label may be exported (by PUBLIC or EXPORT). This function may be called from another module, provided that the label is imported (by EXTERN or IMPORT).

Note that labels and symbols not publicized are not accessible from other modules. These are said to be “local” or “invisible”. Local symbols allow you to hide the private tedious details of modules from the rest of the modules.

Example: The syntax is shown below. Refer to the demo project RelativeAssembly01 in the work directory for a working example.

Module containing serial input/output routines publicize its functions:

```
public  getc                ; function (label)
export  putc                ; function (label)
```

Other modules may call the functions getc and putc, provided that they import the labels.

```
extern  code  getc          ; function (label)
extern  code  putc          ; function (label)
```

14.3.12 PUBLIC (EXPORT)

Function: identify labels or symbols which are publicized to other modules.

Description: PUBLIC is used to specify that the given labels or symbols are defined in the current module and are made available to other modules. EXPORT is an alternative keyword that is interpreted as PUBLIC.

EXTERN (IMPORT) and PUBLIC (EXPORT) pairs provide the primary mechanism for multimodule programming. They allow symbols (variables) or labels (code addresses) to be publicized (exported) by one module and imported (by extern) by another. For example, a function label may be exported (by PUBLIC or EXPORT). This function may be called from another module, provided that the label is imported (by EXTERN or IMPORT). Note that labels and symbols not publicized are not accessible from other modules. These are said to be “local” or “invisible”. Local symbols allow you to hide the private tedious details of modules from the rest of the modules.

Example: The syntax is shown below. Refer to the demo project RelativeAssembly01 in the work directory for a working example.

Module containing serial input/output routines publicize its functions:

```
public  getc                ; function (label)
export  putc                ; function (label)
```

Other modules may call the functions getc and putc, provided that they import the labels.

```
extern  code  getc          ; function (label)
extern  code  putc          ; function (label)
```

14.4 Linker

Currently, the Reads51v4.x linker is configured to generate executable code for Rigel embedded control boards. Please refer to the section Relative Assembly Concepts for a discussion about the linker and for an example.

15 rCHIPSIM51

rChipSim51 simulates the functionality of a standard 8051 in software. That is, it implements a virtual 8051 chip. rChipSim51 may be selected as a target on which the compiled programs run. (Use the “Options | Toolchain / Target Options” menu). rChipSim51 supports the following features:

Standard Interrupts: T0, T1, EX0, EX1, Serial Port (TI+RI)
Standard Timers T0 and T1
Simulated Serial I/O
Simulated Ports

15.1 SimTTY Window and Serial I/O

rChipSim51 supports simple simulated serial input/outputs through the SimTTY window. The simulated serial port need not be initialized, nor the Baud rate generated. The bytes placed in the SFR SBUF are sent to the SimTTY window. Similarly, keystrokes in the SimTTY window are put into SBUF. As in the 8051, SBUF is a double buffer to support concurrent inputs and outputs.

15.2 SimIO Window and Simulated Ports

The functionality of the compiled program may be observed through the SimIO window. The user may interact with the ports while the program is running. rChipSim51 reflects the current state of its four ports (P0 through P3). Note that the 8051 ports have pull-up resistors. Any port may be grounded by simply clicking on the port icon. Such clicks correspond to momentarily grounding the port. That is, the port is grounded as long as the mouse button is held down. Holding the SHIFT key while clicking on a port simulates toggling the port state. The ports in the SimIO window do not show the address, data or control signals when accessing external code or data memory.

APPENDIX A MENU COMMANDS

Menu Item	Hot Key	Action
Project Menu		
New Project		Opens a new project.
Open Project		Opens an existing project.
Save Project		Saves the current project to disk
Save Project As		Saves the current project under a different name
Save Project Copy As		Saves a copy of the current project under a different name
Set Project Active	Ctrl+F10	Sets project as active when more than one project is open.
Project Build Options		
Compiler Options		Opens window to allow you to select compiler options
Assembly Options		Opens window to allow you to select assembly options
Close Project		Close current project
Close All Projects		Close all open projects
Open Workspace		Opens workspace
Save Workspace		Saves workspace
Close Workspace		Close workspace
Recent Workspaces		Shows recently used workspaces
Exit		Exits the program
File Menu		
New File	Ctrl+N	Opens a new file.
Open File	Ctrl+O	Opens an existing file.
Save File	Ctrl+S	Saves the current file to disk.
Save File As		Saves the current file under a different file name.
Save All		Saves all open files
Close File		Closes the current file.
Print	Ctrl+P	Prints the current file.
Print Preview		Displays the page as it will be printed.
Print Setup		Selects printer options.
Module Menu		
Module Properties	Alt+Enter	
Import Module(s)		Imports module from another project
Create Module		Create a new module
Open Module(s)		Open an existing module
Code Wizard	Alt+F10	Not active in this release
Save Module(s)	Ctrl+S	Save current module
Save All modules	Shift+Ctrl+S	Saves all modules
Close Module(s)		Closes module
Delete Module(s)		Deletes module
Cut Module(s)		Cut module onto the clipboard
Copy Module(s)		Copy module onto the clipboard
Paste Module(s)		Paste module from clipboard into a project
Compile Menu		
Build	F9	Compiles or assembles project
Build and Download	Ctrl+F9	Compiles or assembles project and downloads to target
Make Library		Saves .obj files in project to the library
Rebuild All	Shift+Ctrl+F9	Recompiles or reassembles all files
Clean		Deletes all intermediate files of a project.
Toggle BUILD / DEBUG	ModeF2	Toggles between Build and Debug Mode of the IDE
Download Hex		Downloads HEX file to target

Debug Menu

Run to Breakpoint	Ctrl+F8	
Run Skip Breakpoints		
Run to Cursor		
Step Into	F8	
Step Over	Alt+f8	
Step Out	Shift+F8	
Show Next Statement		
Stop Debugging	Ctrl+F2	
Break Execution	Shift+Ctrl+F2	
Restart		
Toggle Breakpoint	F5	Allows you to turn on or off selected breakpoints.
Clear Breakpoint	Ctrl+F5	Removes all breakpoints from your selected program.

Edit Menu

Undo	Ctrl+Z	Restores the document to its state immediately before the last edit command.
Redo	Ctrl+Y	
Cut	Ctrl+X	Cut the highlighted text and places it into the clipboard.
Paste	Ctrl+V	Places the contents of the clipboard into the file at the current carret position.
Copy	Ctrl+C	Copies the highlighted text into the clipboard without removing it from the file.
Select All	Ctrl+A	Selects the contents of the entire file.
Find	Ctrl+F	Finds a string in a file
Find Next	F3	Finds the next instance of the string in the file
Replace	Ctrl+H	Replaces the text with new string
Jump	Ctrl+G	Jumps to the specified code line

View Menu

Toolbar	Toggles on and off the standard toolbar
Status Bar	Toggles on and off the status bar
Workbook Mode	Toggles on and off the workbook mode
Project Manager	Toggles the project window open and closed
TTY Window	Toggles the TTY window open and closed
Output Window	Toggles the output window open and closed
SFR Window	Toggles the SFR window open and closed
Memory Window	Toggles the memory window open and closed

Tools Menu

Find in Files	Find a string in the files
Customize Toolbars	Allows you to customize the toolbar
Burn RIC320 EEPROM	

Options Menu

Toolchain / Target Options	Allows you to set the toolchain and target options
TTY Options	Allows you set the TTY options
Single-File (Projectless) Build Options	
Compiler Options	
Assembly Options	
Editor Options	Allows you to set the editor options for font, syntax highlighting
Environment	
Work Directory	Allows you to set the default work directory
Workbook Icons	Toggles on and off the ICONs on the workbook Tabs
Default Settings	Clears all settings and sets them back to the default settings

Window Menu

New Window	
Cascade	Arranges all editor windows in a cascade fashion

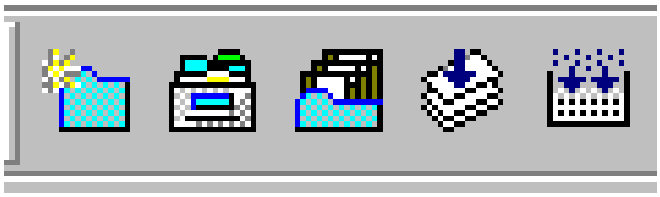
Tile Horizontally		Tiles all editor windows. This is especially useful to view two files simultaneously.
Tile Vertically		Tiles all editor windows. This is especially useful to view two files simultaneously.
Arrange Icons		You may arrange the minimized edit windows neatly by this command.
Close All		Closes all windows

Help Menu

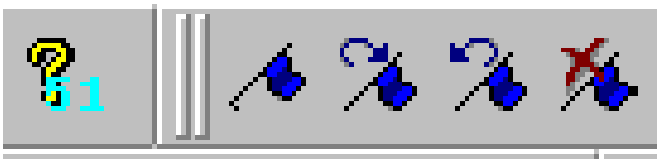
Help Topics	F1	Opens the help files
MCS-51 Overview		Opens the help file for the MCS-51 instructions
About Reads51		

APPENDIX B TOOLBAR BUTTONS

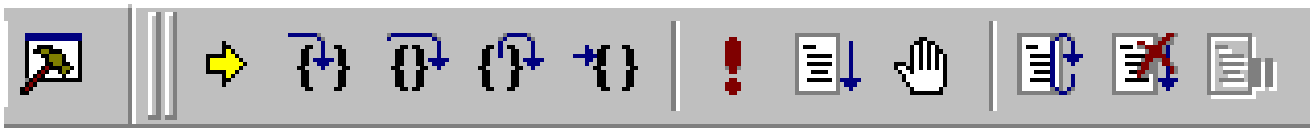
The following are the Toolbar Buttons, which are specific to the Reads51 IDE.



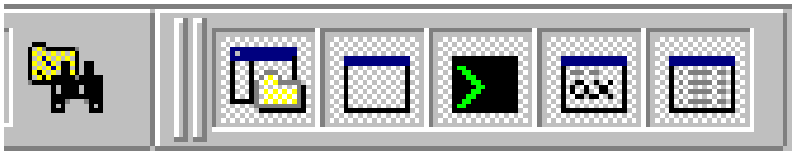
- New Project
- Open Project
- Save Modified Modules (Shift+Ctrl+S)
- Build Active Project (F9)



- Rebuild Project (Shift+Ctrl+F9)
- MCS-51 Help
- Toggle Bookmark
- Next Bookmark
- Previous Bookmark
- Clear All Bookmarks



- Toggle BUILD / DEBUG Mode (F2)
- Show Next Statement
- Step Into (F8)
- Step Over (Alt+F8)
- Step Out (Shift+F8)
- Run to Cursor
- Run Skip Breakpoint
- Run to Breakpoint
- Toggle Breakpoint (F5)
- Restart
- Restart Program Execution
- Stop Program Execution



- Find String in Multiple Files
- Project
- Output
- TTY
- SFR
- Memory Page

APPENDIX C RELATIVE ASSEMBLY CONCEPTS

Relative assembly is sometimes referred to as relocatable assembly. Mechanically speaking, it provides the basis for modular programming, be it assembly or any high-level language (HLL) such as C. The entire software is regarded as a collection of modules. Parenthetically, terms such as module and segment are frequently used in conjunction with software and may mean different things in different contexts.
Each such software module is considered as a building block. Each module may have locally used variables, invisible to other modules. Eventually, modules must interact. For example, a function (subroutine) in one module may be called from another module. In this case, the function address (code label) need be publicized by the module. Similarly, a module may contain data that needs to be accessed by other modules. In this sense, such data is no longer local and invisible to other modules. Rather, it is global data. Accordingly, modules that contain global data must make the data addresses (labels) public. Code or data defined in other modules are said to be external references to the module which needs access to them. Clearly, when such a module is assembled (by a relative assembler) the result is not readily executable. That is because the exact value (address) of external references is not known. A closer inspection of a module reveals that modules contain different types of labels and symbols, primarily depending on their place in the memory map, or memory space. For example, labels to code and labels to external data need to be differentiated in the 8051. This also implies a block of code memory, most probably containing machine instructions, must be treated differently from a block of external data memory, perhaps containing global variables. Relative assembly takes this distinction a step further: more than a single block of a given type of memory may be defined as a cohesive unit. Such units are called segments. Again, the term segment may be somewhat confusing to the first-time users, since the same term is used for the collection of all segments of the same type. We will clarify this later after we discuss the linker. In the MCS-51 architecture, a module may contain one or more segments of type code, external data, internal direct data, internal indirect data, or bit. The keywords CODE, XDATA, DATA, IDATA, and BIT are used to designate these types.

The output of the relative assembler is referred to as an object module. Object modules are usually composed in binary. They contain the output code from the relative assembler, but lack any spacific address information. For example, branches to absolute addresses are not completely specified. Instead, the object modules contain the so-called fixup records. Fixup records list the function and symbols made public by the current module, as well as external references needed to generate executable code.

Modules of an application are all assembled, yielding a set of object modules. The term “relative” in relative assembly comes from the fact that any absolute start address (also called offset or base address) may be assigned to the module. The term “relocatable” also implies this aspect. Once a set of object modules are at hand, the final step is called linking, and the program that performs this is called a linker. The linker takes the object modules, reviews the public labels and symbols as well as the external references. Several checks are performed. For instance, if a module specifies an external reference, say a function label, but none of the modules have publicized the function label, it becomes impossible to generate executable code. This is often referred to as the module having “unresolved external references.” Similarly, an ambiguity arises if the same function (label) is publicized by more than one module. In such a case, it is not clear which function should actually be called. If no such inconsistency is detected, the linker proceeds by collecting the modules into one executable program.

Object modules are stacked following precise rules. Typically, first all of the segments of the same name of each module are stacked together. Then all the modules are scanned and all the segments of the same type are stacked, keeping the segments with the same name as contiguous blocks. This is done for all five segment types of the MCS-51 architecture. At the end of this aggregation the total size of each segment type is known. Moreover, the offset of each segment of each module is known. Usually some size checking is performed to verify that the segments would fit into the available resources. Finally, the linker locates the segments. That is, absolute starting addresses are assigned to each segment. Since locating the segments is a fundamental task in generating final executable code, the linker is sometimes referred to as a linker/locator. At a minimum, the start address of code and external data memory need to be specified. Once known, the linker may now compute the absolute address of each segment. This information is subsequently used, along with the fixup information contained in the object modules, to resolve all external references and all absolute internal references. This approach to generating executable code is fairly flexible. In fact, almost all approaches to assembly language programming are supported as special cases. Moreover, keywords and pseudo ops are provided to support absolute assembly.

For example, it is possible to specify an absolute origin to a module. If the module does not have external references, then the assembler may generate executable code, just as an absolute assembler. Similarly, code may consist of a single module with only one segment for each type. In this case, linker simply stacks the segment types and locates the code into an executable program. On the other hand, a HLL may take advantage of the features of relative assembly and the multi-module programming support it provides. For instance, the 'C' language keyword extern is simply forwarded to the relative assembler, specifying the variable to be defined in another module.

Example

```
; a minimal two-module relative assembly source
; -------------------------------------------------------------
; module 1
; -------------------------------------------------------------

Routines    segment code
Variables   segment xdata

; imported labels and symbols (defined in other
; modules but referred to in this module)
extern    code    putc       ; function (label)
extern    xdata   Ch         ; external (RAM) data (symbol)

; code written to an absolute code segment at the reset vector
; the following code is automatically added by the project
; manager it assumes there is a function called "main"

cseg at 0                    ; cseg is the keyword to start an absolute
                             ; code segment
ljmp    main
end                          ; each segment must terminate with an"end"
                             ; directive

; code written to the relative code segment "Routines"
rseg    Routines      ; rseg is the keyword to start a
                             ; relative segment

main:                        ; this label is exported
mov     dptr, #Ch            ; address of variable Ch
movx    a, @dptr             ; get Ch
lcall   putc                 ; putc prints Ch (in acc)
ret                          ; done
end                          ; each segment must terminate with an
                             ; "end" directive

; -------------------------------------------------------------
; module 2
; -------------------------------------------------------------
Routines    segment code
Variables   segment xdata

; declare exported labels and symbols (defined in this
; module and referred to in other modules)
public    putc               ; function (label)
public    Ch                 ; external (RAM) data (symbol)

rseg    Variables            ; rseg is the keyword to start a
                             ; relative segment
Ch:                          ; this label is exported
ds 1                         ; reserve 1 byte for variable Ch
end                          ; each segment must terminate with an
                             ; "end" directive
```

```
rseg Routines                ; rseg is the keyword to start a
                             ; relative segment
putc:                        ; this label is exported
clr  TI                      ; transmit flag
mov  sbuf, a                 ; send char in acc
jnb  TI, $                   ; $ has the value of the current
                             ; location pointer
                             ; i.e., the current address
clr  TI                      ; transmit flag
ret
end                          ; each segment must terminate with an
                             ; "end" directive
```

APPENDIX D A BRIEF REVIEW OF C

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.

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.

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.

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.

Variables

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.

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;
```

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]:

Constants

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.

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.

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;
        }
```

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.

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.

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.

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
       }
```

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.

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.

The compiler is not an ANSI C compiler. It is written with a graphical Integrated Development System (IDE) in mind. The compiler 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 the compiler the “Hello” 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.

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

APPENDIX E SmallC

SmallC implements a subset of the K&R C language. It was written by Ron Cain and published in the May 1980 issue of Dr.Dobb's Journal. Later, James E.Hendrix improved and extended the original SmallC compiler. He describes the SmallC compiler in the book "The Small-C Handbook", ISBN 0-8359-7012-4 (1984). Originally, SmallC was written to produce 8080 assembly language code from the C source.
Since its introduction, it has been ported to several processor and microcontrollers. Many of these implementations are in the public domain. Consequently, SmallC has been a popular choice of experimenters, educational institutions and embedded systems developers.
It has a few restrictions compared to K&R C or ANSI C:

1. Structures and Unions are not implemented
2. Only one-dimensional arrays are allowed.
3. Only one level of indirection (pointer) is allowed.
4. Only integer and character types are allowed.

The C compiler in Reads51 is a SmallC-compatible compiler that generates MCS-51 relative assembly language from C source. The output is intended to be assembled by the Reads51v4 relative assembler and subsequently linked by the Reads51v4 linker.
The C compiler has some of the limitations of SmallC. However, it also introduces some significant extensions and improvements over standard SmallC.

Reads51v4 C Compiler:

1. Uses the more modern (ANCI C) function argument definition syntax.
2. Arguments are passed to the functions in the C convention. This allows a variable number of arguments to be passed on to functions, such as in printf().
3. Supports MCS-51 interrupts.
4. Supports function prototypes.
5. Supports the void type.
6. Uses Rigel's proprietary macro preprocessor.
7. Supports sfr and sfr bit types.

APPENDIX FOVERVIEW OF THE MCS-51 INSTRUCTION SET

MCS-51 Addressing Modes and Notation

The addressing mode refers to the various ways operands are specified. For example, move instructions require a source and a destination, or addition requires two operands.

The MCS-51 Instruction Set

Instruction	Function
ACALL	addr11 absolute call
ADD A,<src-byte>	ADD adds a source byte to the accumulator.
ADDC A,<src-byte>	ADDC adds a source byte to the accumulator with carry.
AJMP addr11	Absolute jump
ANL <dest-byte>,<src-byte>	Logical AND for byte variables
ANL C,<src-bit>	Logical AND for bit variables
CJNE <dest-byte>,<src-byte>,rel	Compare and jump if not equal
CLR A	Clear accumulator
CLR bit	Clear a bit
CPL A	Compliment accumulator
CPL bit	Compliment accumulator
DA A	Decimal adjust accumulator for addition
DEC byte	Decrement byte
DIV AB	Divide
DJNZ <byte>,<rel-addr>	Decrement byte and jump if not zero
INC byte	Increment byte
INC DPTR	Increment data pointer
JB bit,rel	Jump if bit set
JBC bit,rel	Jump if bit set and clear bit
JC rel	Jump if Carry is set
JMP @A+DPTR	Indexed jump
JNB bit,rel	Jump if bit not set
JNC rel	Jump if Carry is not set
JNZ rel	Jump if accumulator is not zero
JZ rel	Jump if accumulator is zero
LCALL addr16	Long Call
LJMP addr16	Long Jump
MOV <dest-byte>,<src-byte>	Move byte variable
MOV <dest-bit>,<src-bit>	Move bit data
MOV DPTR,#data16	Load data pointer with a 16-bit constant
MOVC A,@A+<base reg>	Move code byte
MOVX <dest-byte>,<src-byte>	External move.
MUL AB	Multiply
NOP	No operation
ORL <dest-byte>,<src-byte>	Logical-OR for byte variables
ORL C,<src-byte>	Logical-OR the Carry Bit with a bit variable.
POP direct	Pop from stack
PUSH direct	Push onto the stack
RET	Return from subroutine
RETI	Return from interrupt
RL A	Rotate accumulator left
RLC A	Rotate accumulator left the Carry flag
RR A	Rotate accumulator right
RRC A	Rotate accumulator right through Carry flag
SETB <bit>	Set bit
SJMP rel	Short jump
SUBB A,<src-byte>	Subtract with borrow

SWAP A	Swap the two Accumulator nibbles.
XCH A,<byte>	Exchange Accumulator with byte variable
XCHD A,@Ri	Exchange digit.
XRL <dest-byte>,<src-byte>	Logical Exclusive-OR for byte variables

APPENDIX G OMF-51

The OMF-51 (“Object Module Format for the MCS-51) was developed by Intel. It has become the de facto object file standard for the MCS-51 language. Almost all professional assemblers, compilers, and in-circuit emulators (ICEs) support the OMF-51 specifications. The specifications are freely available on the Intel web site as well as other web sites. Refer to the Rigel Corporation web site www.rigelcorp.com Download Documents to find a copy in PDF format.

APPENDIX H R-515JC BILL OF MATERIALS

The bill of materials given below lists all components by their reference as they appear on the board top overlay.
Revised: MAY1999
List Of Materials

Item	Quantity	Part	Reference	Description
	CAPACATORS			
1	19	10nF AXIAL	C1, C14-C31	Axial capacitor
2	12	1uF sm	C2-C13	Electrolytic capacitor
3	2	47uF 16V	C33, C34	Electrolytic capacitor
4	1	100uF 16V	C32	Electrolytic capacitor
	RESISTORS			
5	1	10K 8 Gang	R6	Gang Resistor
6	3	10K 10Gang	R1-R3	Gang Resistor
7	2	330 OHM 1/2W 5%	R4, R5	½ Watt 5% Carbon Resistor
8	1	1K 1/4W 5%	R7	¼ Watt 5% Carbon Resistor
9	2	10K 1/4W 5%	R9,R10	¼ Watt 5% Carbon Resistor
10	1	20K FLAT POT	R8	Flat 6mm Potentiometer
	DIODES			
11	1	LG Red LED	D1	LED T1 ¾ (red)
12	1	LG Green LED	D2	LED T1 ¾ (green)
13	2	1N4001	D3, D4	Diode
	CONNECTORS			
14	2	DB9 FEMALE	P1, P2	DB9 Connectors
15	1	2X 32	J20	.100 2X 32 Header
16	1	1X 32	J20	.100 1X 32 Header
17	3	2 Position Small TB	J18, J7	Terminal blocks (14)
18	4	3 Position Small TB	J19	Terminal block (2)
19	1	Battery Holder	B1	Battery Holder
20	1	Pansonic Battery	BR1225	Battery
21	1	Push Button	PB1	6 mm Pushbutton
22	1	Slide Switch	S1	Slide Switch
23	3	1X 5 Shrouded	J10, J11, J12	1X 5 Shrouded Header
24	3	1X 6 Shrouded	J13, J16, J17	1X 6 Shrouded Header
25	3	1X 3 headers	S4, S5	.100 1X 3 Headers
26	2	1X 2 Header	J21, J22	.100 1X 2 Headers
27	1	1X 6 Header	J4	.100 1X 2 Headers
28	1	1X 7	J14	.100 1X 7 Headers
29	1	1X 8	J15	.100 1X 8 Headers
30	1	2 X 11	J1/J2/J3/J5/J6/J8/ J9	.100 2X 11 Headers
31	1	2X 3	S2/S3	.100 2X 3 Headers
	SOCKETS			
32	3	8 DIP	U3, U8, U14,	8 Pin Dip Socket
33	2	14 DIP	U15, U16	14 Pin Dip Socket
34	3	16 DIP	U10, U11, U12	16 Pin Dip Socket
35	1	20 DIP	U1	20 Pin Dip Socket
36	1	24 DIP (WIDE)	U17	24 Pin Dip Socket
37	2	32 DIP	U6, U7	32 Pin Dip Socket
38	2	PLCC44	U4, U9	44 Pin PLCC Socket
	ICS			
39	1	XC9536-15PC44C	U4	PLD
40	1	74HC573	U1	Octal latch

41	1	XR88C92CJ / XR88C92IJ	U9	Dual UART chip
42	1	27C256 / 27C512	U7	32K EPROM
43	1	62256 / 68512	U6	32K Static RAM
44	1	SAB C515C-LM	U2	Microcontroller
45	1	PCA 82C250	U13	CAN Interface IC
46	3	MAX232	U10,U11,U12	RS-232 IC
47	1	DS1233-10	U5	Reset Chip
48	1	DS1218	U8	Memory Battery Back-up IC
49	1	DS1685-5	U17	Real Time Clock
50	2	MAX491CPD	U15, U16	RS422/485 IC
51	1	32.768kHz	Y1	Crystal
52	1	10MHz Clock	U14	Microcontroller Clock
53	1	3.6864MHz Clock	U3	UART Clock

APPENDIX I TOP OVERLAY