RIDE

Part 2: Debugging

80C51 and 80C51XA
Development Tools

January 2000

RAISONANCE

The information included in this manual may be modified by RAISONANCE S.A and is in no way a commitment of RAISONANCE S.A. The rights of use of the software described in this document may only be transferred under a licensed agreement. Thus, the software may only be used under the terms of this agreement. All copying of this document for any other purpose besides the personal use of the buyer, must have prior authorisation from RAISONANCE SA.

The present manual covers the RIDE software.

Copyright 2000 RAISONANCE SA. All rights reserved.

REGISTERED TRADEMARKS:

RAISONANCE is a registered trademark of RAISONANCE S.A.

CONTENTS

1. Introduction

1.1 Presentation

1.2 Installation

1.3 About the manual

1.4 RIDE for which tools ?

1.1 Presentation

RIDE is a Windows integrated development environment which contains a set of development tools for 8051 and XA microcontroller applications.

This development environment includes:

- a color syntax highlighting editor,
- a project manager,
- coding tools: Assembler, C Compiler,
- utilities: linker, library manager,
- Simulator/ Debugger interface.

1.2 Installation

To install RKIT, insert the CD-ROM and follow the instructions. If it is not automatic, select first the directory corresponding to the language you wish to use (in the root of the CD). Then, run **INSTALL.EXE** from this directory.

Follow the information of the installation procedure (see **readme.txt**)

1.3 About the manual

The RIDE documentation is divided in 2 parts:

- **RIDE Part 1: Coding** describes the editor, and the associated coding tools.
- **RIDE Part 2: Debugging** describes the Debugging interface, for the simulator.

The following documentation contains 9 chapters:

1. **Introduction**
2. **Working with the debugger**
3. **Visualizations of the application**
4. **Execution control**
5. **Trace features**
6. **Bank Switching**
7. **Options summary**
8. **Commands summary**
9. **Index**

1.4 RIDE for which tools ?

RIDE, is a 32-bit version and is available for Windows 95, 98 and NT.

The Windows versions of RKit compiler, assembler and simulator are Dynamic Link Library files « .DLL » which could be used from **RIDE**.

RIDE could be configured to work with a changing number of tools. In general, the initial configuration is set by the installation process and the serial number of the user will determine the tools that could be fully used and the ones that are usable as evaluation versions.

The kits available are :

- **RKit51** for the 8-bit 80C51 family, containing the following tools :
  1. ANSI-C RC-51 compiler, with its libraries
  2. MA-51 macro assembler
  3. LX-51 linker
  4. KR-51 kernel
  5. integrated debugger (Simulator and Emulator)

- **RKitXA** for the 16-bit 80C51XA family, containing the following tools :
  1. ANSI-C RC-XA compiler, with its libraries
  2. MA-XA macro assembler
  3. RL-XA linker
  4. KR-XA kernel
  5. integrated debugger (Simulator)

RKit51&XA containing both of the above

2. Working with the debugger

2.1 Invocation of the debugger

2.2 Configuration of the debugging session

2.3 Debugging an application

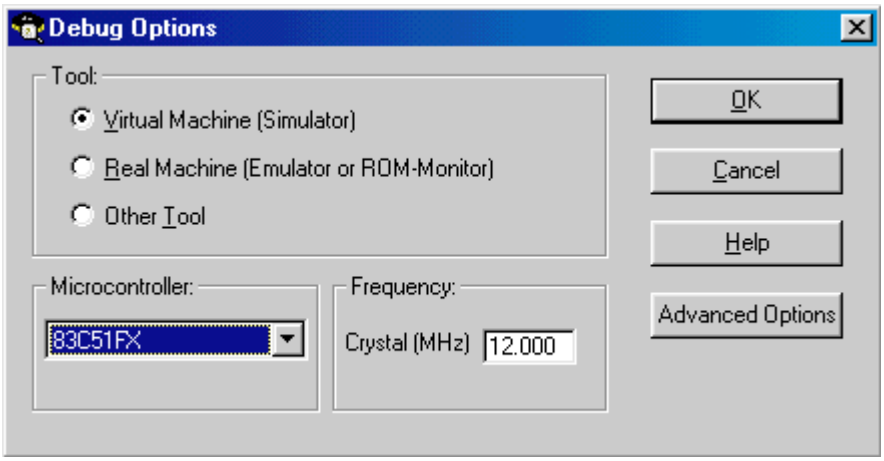
2.1 Invocation of the debugger

The installation process will create the RKIT program group, and the following short-cuts:

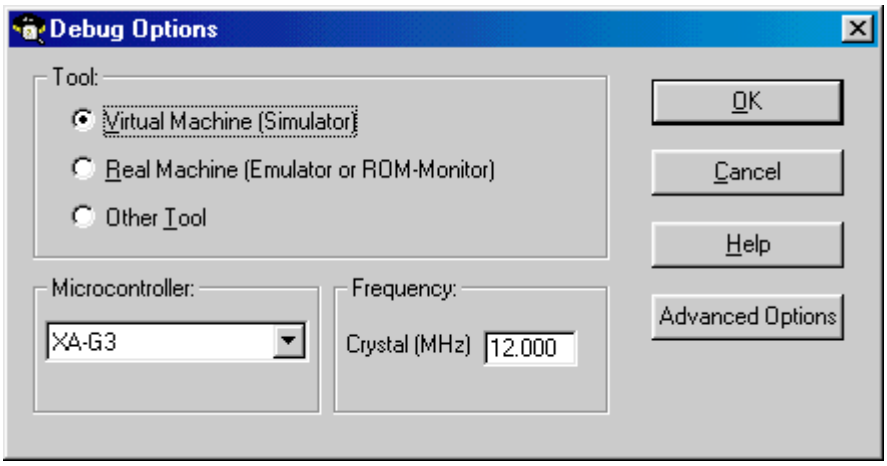
1. the **Ride I.D.E** short-cut allows to run the Integrated Environment Interface, for RAISONANCE encoders (RC-51, RC-XA, MA-51, MA-XA) and the integrated debugger. The full command is: RIDE.EXE.
2. The **Ride Reference** short-cut will open the One-line Help.

2.2 Configuration of the debugging session

Before any debugging session, you have to select **Options|Debug** to define your application options. The following dialog box appears:



8051



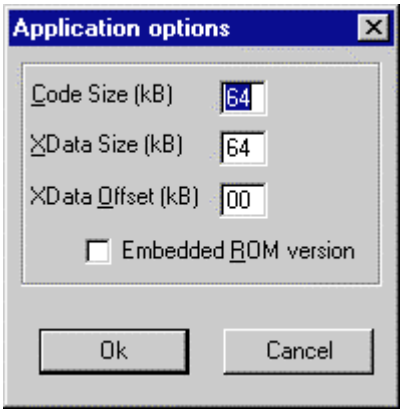
80C51XA

The first choice concerns the selection of the debugging tool: **Virtual Machine** (Simulator) or **Real Machine** (Emulator). The Other tool selection is useful for external DOS/ Windows debugging tools. To run the other tools from RIDE, see the chapter « Other debugging tools ».

Note: The choice of the microcontroller has to be made in the **Options|Target** window

To configure the simulator,

- Select **Virtual Machine**,
- Select the **Microcontroller** you want to simulate
- Enter the clock frequency in the **Crystal (MHz)** window.
- Open the **Advanced options** window. This box is different depending on the microcontroller family you simulate.



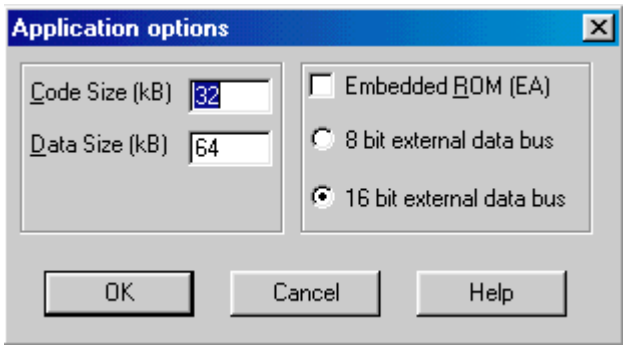
8051

■ **Code size** : Code size of your application (in Kbytes). You can always keep the maximum size 64 KB, even for smaller applications. For Bank switching mode, keep always the value 64 KB.

■ **Xdata Size** : XDATA size of your application (in Kbytes). For Embedded ROM application, where you have no XDATA, please enter 0 KB.

■ **Xdata Offset** : XDATA offset (in Kbytes) corresponding to the first XDATA address, of your application.

■ **Embedded ROM version** : Click this choice for embedded ROM applications.



80C51XA

■ **Code size** : Code size of your application (in Kbytes). You can always keep the maximum size 32 KB, even for smaller applications.

■ **Data Size** : DATA size of your application (in Kbytes).

■ **Embedded ROM version** : Click this choice for embedded ROM applications.

■ **8 bit external data bus** :.

■ **16 bit external data bus** : .

2.3 Debugging an application

The debugging is made with the OMF-51 or OMF-XA generated by the linker.

To be able to start a debugging session, you have to open an existing project, or create one.

First of all, before starting the debugging, it is necessary to configure the debugger, with the **Options|Debug** command.

2.3.1 Starting the debugging mode

To start a debugging session, choose the command **Start** in the **Debug** menu.

Note: This command can also start the tool chosen with Other tools option.

The **Debug|Start** command initializes the program and begins a debugging session. If one of the projects file have been changed since the last « make » of the project, RIDE will « make » the project before starting the debug. If the Debug options are not set, RIDE displays the **Options for new CPU** dialog box where you can set these options.

Note: Debugging mode will work properly only with a .AOF file with DEBUG information. These Debug information (Lines, Symbols, ETC..) are set in the Compiler, assembler and Linker options (see Options|Compiler, Options|Assembler and Options|Linker).

When the command **Debug|Start** has been completed, the specific Debug commands become available and the execution point (displayed with a highlighted blue line) goes to the beginning of the program.

2.3.2 Closing the debugging mode

To close the debugging session and come back into RIDE, choose the command **Debug|Terminate**.

This command will close the debugging windows but not the source windows

3. Visualizations of the application

3.1 Windows management

3.2 Symbolic debugging

3.3 Variable space views

3.4 Other views

3.1 Windows management

Several commands allow the debugging windows to be tiled.

3.1.1 Saving the context

Between 2 debugging sessions, the debugging windows organization is saved.

So, each time you enter in the debugger, the debugging windows opened in the last session are opened at the same place and with the same size.

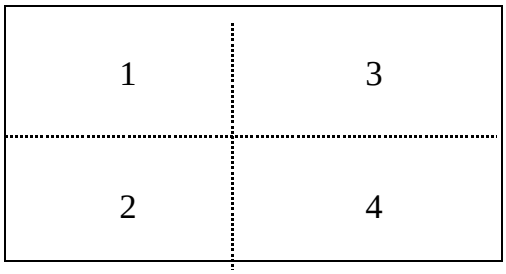
3.1.2 Manual tiling

As any Windows application, the debugging windows can be moved or resized manually.

3.1.3 Automatic tiling.

The **Windows|Tile Debug** command organizes the open windows of your actual debugging session.

The open windows will be divided in 4 groups:



1. This group remains the Code windows (sources and code)
2. This group remains main registers and peripherals
3. This group remains the memory spaces
4. This group remains the watch and the trace windows.

3.1.4 Tile bars

The **Tile Bars** command organizes the Debug Windows. This command makes visible 2 cursors (Left/Right and Up/Down) which define a Source zone where is the Source and Code window. You can increase or reduce this zone with these 2 cursors.

3.1.5 Refreshing the open data windows

The open data windows are refreshed at each break of the program (after a break-point or a step by step command). Moreover, it is possible to refresh the data during the execution:

The Windows|Refresh command refreshes the open data windows (as well as watched data) during the execution of the program. This command will replace the animating push-button if you don’t need to refresh the values all the time.

With the emulator, the data windows are refreshed only after a break during the program.

The exception are the xdata dump window, if it is declared in emulation RAM.

Program Windows

To visualize the program, 2 views are available:

1. The source editing windows.
2. The code windows, which are the result of the disassembling of the program.

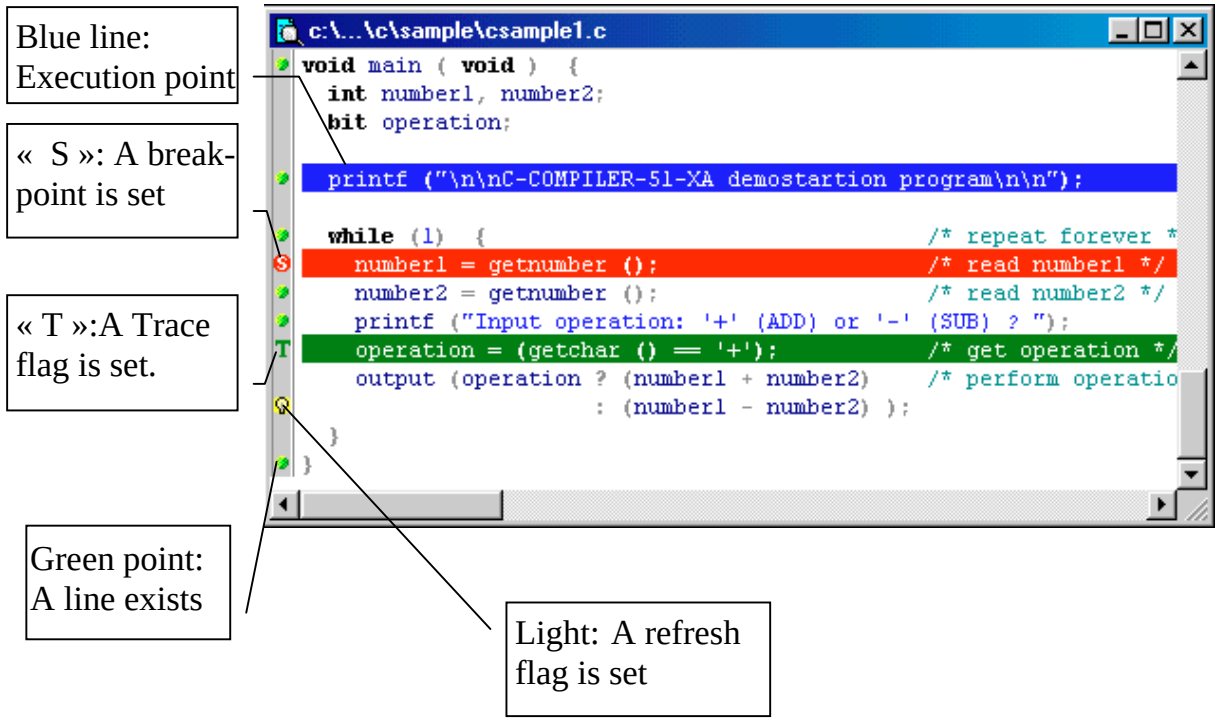
3.1.6 Execution point.

The **Debug|Show execution point** is available under the debug mode when the program is not running. It highlights the next line to be executed. This command is available for both Code and source windows.

3.1.7 Editing windows (Source).

The source window where is the execution point is automatically opened.

Some information are displayed at the left side (control flags, active lines):

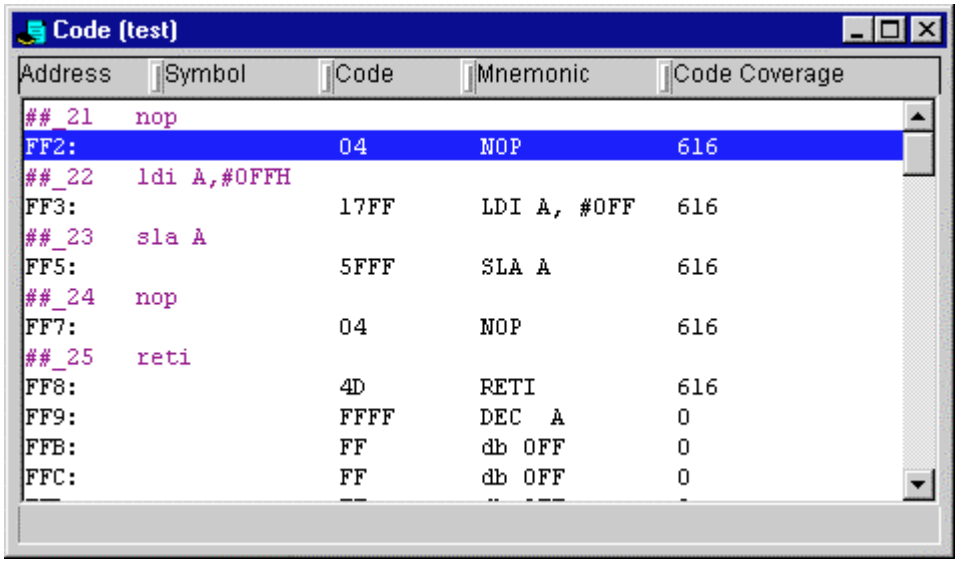


In addition, you can access a popup command menu (click on the mouse right button).

3.1.8 Code view

The **Debug|Code** command opens the Code window.

It displays the direct generated machine code (in Hexadecimal) and the corresponding assembler mnemonic:



- The source line is displayed in front of the generated code.
- Lines numbers (for the source) and the corresponding Hexadecimal absolute address are displayed.
- You can modify the generated code in this window, with a double-click in the symbol, code or mnemonic.
- A POPUP menu (mouse right button) allows you to:

1. **View source**: This command opens the source file at the line corresponding to the selected address.

2. **Save code**: This command lets the user save the Code content in a file with the format of his choice. The available formats are: Source, Binary, Hex Intel.

3. **Set symbol**: To associate a symbol to an address select the corresponding address and choose **Set symbol** in the popup menu of the window. RIDE displays a dialog box where you can type the name of the symbol.

4. **Search|Goto address**: To move quickly to a specific address, choose **Go to address** from the popup menu. RIDE displays a dialog box where you can type the address you want to reach.

5. **Search|Search symbol:** This command displays the **Search** dialog box where the user can type the symbol to search and select the address.

6. **Fill segment:** This command allows to initialize a code area with the same value.

7. **Save segment** : This command allows to save a code area into a file .

8. **Explore**: In some cases, the code could be not disassembled or «explored», specially, with unexpected events, such as indirect jump instructions and conflicts. To resolve this problem, please use Explore in the popup menu then it will «explore» the code from the cursor position. See the next chapter.

9. **Source lines:** This command inserts the source lines of the corresponding address in the code view.

10. **Toggle flag**: This command lets the user set or suppress execution control flags at the selected address:

    - **Toggle** (/remove) **breakpoint:** This command sets/ removes a breakpoint at the cursor location of the active Edit window. The breakpoint is symbolized with a red highlighting on the line to break. When the program stops on the line to break, the highlighting becomes pink.

    - **Toggle** (/remove) **trace**: This command inserts/removes a trace flag to the currently selected source or code instruction in the active Edit Window or Code Window. If the currently selected line doesn't correspond to a valid instruction the flag will be set to the next valid address. The corresponding line is marked by **T**.

    - **Toggle Refresh :** This command sets a refresh flag. The opened data windows will be refreshed when the program executes the flagged line.

    - **Toggle switch:** This commands is useful for a bank switching program. See the Chapter 6 « Bank switching ».

3.1.9 Exploration principles

The « exploration » (or disassembly) is an automatic operation when the Object file is loaded into the debugger. The result of the exploration is displayed in the code window.

3.1.9.1 Intelligent disassembly

The process is to « explore » the code to locate the first byte of each instruction. Each execution thread is followed, starting from all possible "entry points" which may be:

- the address 0,
- the addresses of the interrupt vectors (corresponding to the microcontroller being emulated or simulated),
- the addresses of Line numbers (for high-level language programs).

Bytes of executable code explored in this way are then symbolically disassembled. Bytes not on any execution thread are considered to be "data" or "messages" contained in ROM. These bytes are displayed in hexadecimal and (if appropriate) ASCII.

3.1.9.2 Incomplete exploration

In some cases, the exploration may be stopped by unexpected events, such as indirect jump instructions and conflicts:

Causes

The indirect jump instructions such as 'JMP @A+DPTR', or a 'PUSH DPL, PUSH DPH, RET' sequence, cannot be explored initially. Each time the exploration process finds an instruction such as 'JMP @A+DPTR', a warning message is generated indicating the exact address.

Effects

A zone that has not been explored does not appear as disassembled, but as if it were In-ROM constant data. However, this does not affect execution.

Remedies

The explore command of the code window popup menu allows to disassemble the non-explored areas to be disassembled.

☞ *A program generated by a compiler is totally explored if the complete line-numbers table has been loaded.*

3.1.9.3 Conflictual exploration

Causes

There is a conflict error every time the exploration process finds a byte claimed by two distinct instructions (e.g. a 'JMP' instruction towards the second byte of an instruction already analyzed). There may be three sources of error, a linker error, bad manual address handling, or most often, from an exploration process starting at an incorrect address (non-used interrupt). The conflict error could even be deliberately caused by the end-user (for example, in order to get a high-density code optimization).

Other origins of conflicts are 'particular' handling of the microcontroller stack used by some Compilers, to optimize the generated code. For example, for the assignment of variables in floating point such as:

var = 3.14116;

The Compiler will place the bytes corresponding to 3.14159 (IEEE Format) just after the LCALL towards the function causing the assignment.

Exploration of the corresponding generated code inevitably leads to conflict upon tree-like exploration. In this case, we suggest that you ignore these conflicts.

Effects

Every conflict error stops the exploration process. In addition, execution permanently compares the executed code coherence to the exploration result, and stops at each anomaly to indicate it.

Remedies

Ignore the conflict error, and continue the exploration process.

3.2 Symbolic debugging

When RIDE loads format files such as OMF-51 or OMF-XA, you can watch the variables from the symbol they are attached to. With RC-51 or RC-XA, you also can get Type information, and watch complexes variables such as structures, array, etc..

3.2.1 Expression evaluation

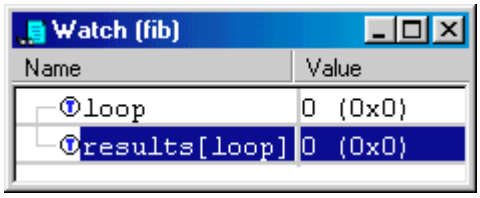
The **Debug|Modify** command displays the **Evaluate expression** dialog box. It lets the user evaluate and modify the value of simple typed variables:

1. Type the expression in the **Expression** edit control.
2. Select the **Evaluate** button to evaluate the expression.
3. Type the new value of the expression in **New value** control.
4. Select the **Modify** button to change the expression value.

3.2.2 The Watch window

The **Debug|Watch** command displays the **Watch** window.

The Watch window lets you monitor the value of a variable or expression during a debugging session. The Watch window displays the current value of the watch expression based on the scope of the execution point:



- The Watch window is blank if you have not added any watches.
- The symbol values are given in Hexadecimal.
- You can open a POPUP menu (right button of your mouse) to **edit**, **format**, **add** and **delete** a watched expression.
- The " " symbol indicates that the watched variable can be visualized in the trace window, click on.

3.2.3 Adding a new expression to watch

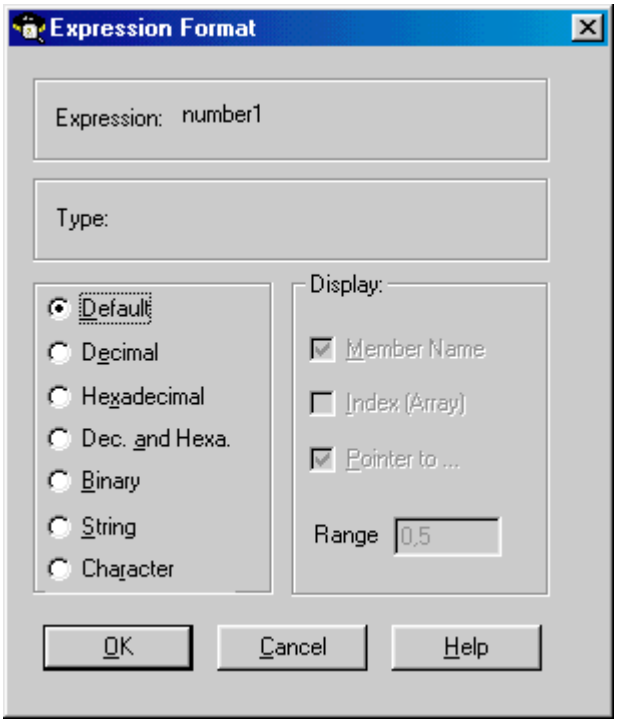
The symbols to be watched can be inserted in the **Watch** window by different ways:

- With the popup menu of the **Watch** window (see next paragraph)
- With the command **Insert watch** of the **Global symbols** window (**Debug**|**Symbols** command)
- With the **Debug**|**Add watch**
- With the button named **Add a new watch expression:** 

3.2.4 Expressions format

From the Watch window, you can change the expressions format:

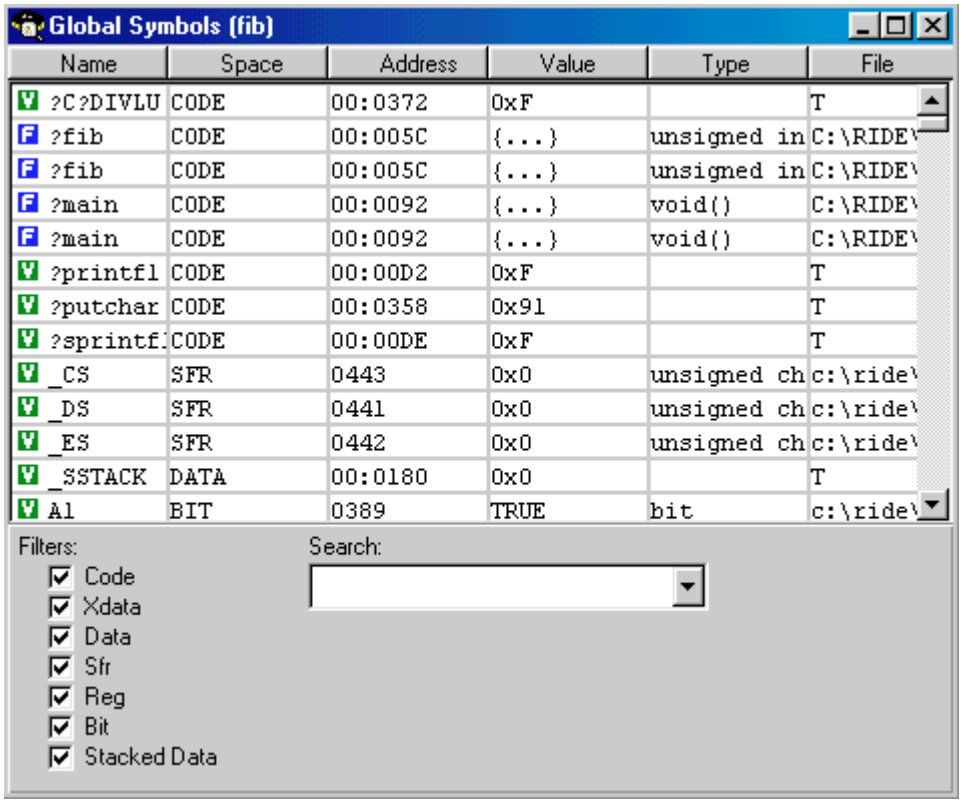
- Select the variable to format
- Click on the mouse right button: The popup menu appears.
- Choose the format command: The **Format expression** windows appears:



- You can display the variable with the following formats: **Decimal**, **Hexadecimal**, **Decimal and Hexadecimal binary, string** or **character.**
- For the structures, you can display the **member name**.
- For the arrays, you can display the **index**, the displayed **range** (ex: 0,5 will display the 5 first elements).

3.2.5 Symbols and expressions list

The global symbols of your program can be listed with the command **DEBUG|SYMBOLS:**

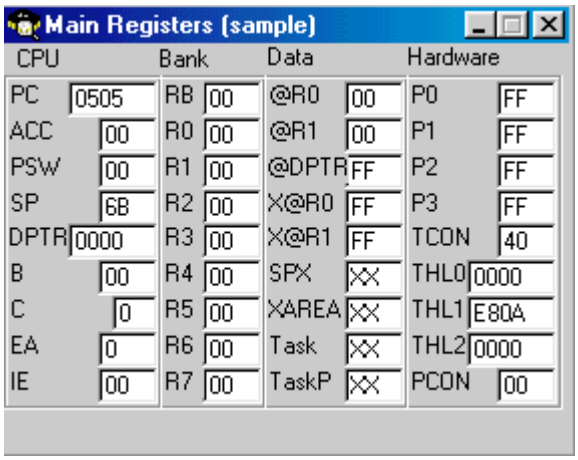


- The list is sorted in alphabetic order (**by name**) but it also can be sorted **by value**.
- Filters (Data, Xdata, Sfr, Bit, Code) , Reg(XA only), Stacked Data) (XA only) allow to be displayed (or not) the symbols from different spaces.
- Each symbol is described: Space, Type (for a C function), address (in Hexa), value (in Hexa) are displayed for the highlighted symbol.
- **Add Watch** allows the highlighted symbol to be entered in the Watch window.

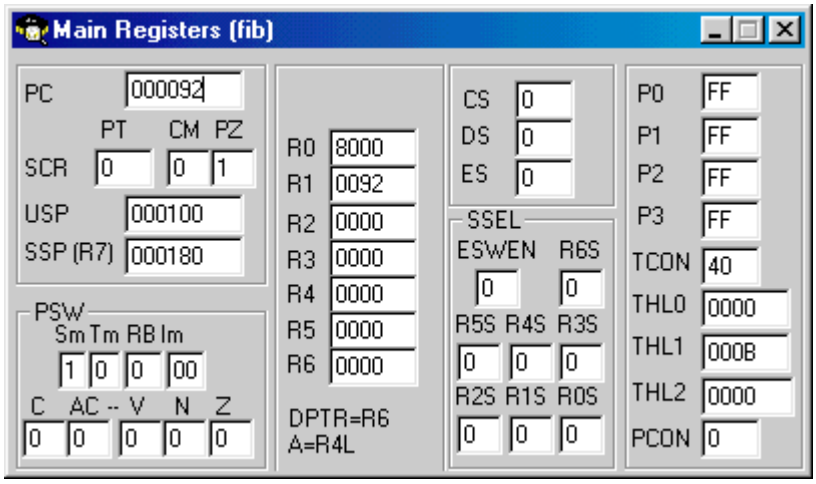
3.3 Variable space views

3.3.1 Visualization. of the Main registers

The **Debug|Main registers** command displays the main registers of the selected microcontroller. In this window, you can also modify the value of each register:



8051



80C51XA

Note: During running the program, each value is refreshed only in the **animating mode** or with the command **Refresh.**

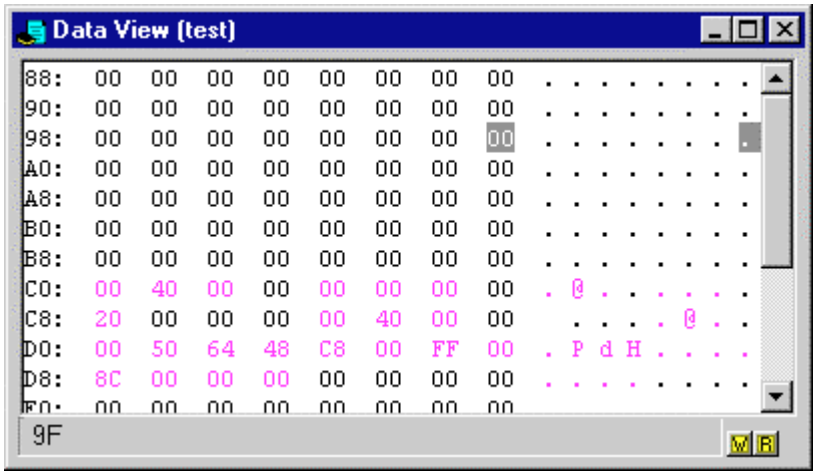
3.3.2 Visualization of the memory

The memory, in an application based on microcontrollers, can be 4 types :

- XDATA : External memory.
- DATA: Internal data memory.
- SFR. : Registers.
- REG**Erreur! Signet non défini.** : Registers
- BIT **Erreur! Signet non défini.**: Bit-addressable registers.
- STACKED DATA : Stack

To visualize one of these 4 memory types, please open the windows :

- **XDATA** : with the **Debug|Memory|Xdata view** command
- **DATA** : with the **Debug|Memory|Data view** command
- **SFR** : with the **Debug|Memory|Sfr view** command
- **REG** : with the **Debug|Memory|Reg view** command
- **BIT** : with the **Debug|Memory|Bit view** command
- **STACKED DATA** : with the **Debug|Memory|Stacked Data view** command



- The window titles the memory type, and the current project file.
- From left to right, are displayed the address, the value in hexa, the value in ASCII. The display is organized in 8 columns (default mode).
- The active address value and its corresponding ASCII character is highlighted.
- Addresses associated with a symbol are displayed in a specific color.
- The active address and the eventual associated symbol is displayed in the window's status bar.

■ For changing the value of a variable, double click the corresponding address in the active window. RIDE displays an edit window to get the new value. To validate this new value press Enter.

■ a popup menu allows:

1. **Setting a symbol**: To associate a symbol to an address, select the corresponding address and choose **Set symbol** in the popup menu of the window. RIDE displays a dialog box where you can type the name of the symbol.

2. **Changing the columns number:** You can select the number of columns you wish to display in the window. To do so, click the right mouse button to display the corresponding popup menu, where you can select **Number of columns.**

3. **Go to address:** To move quickly to a specific address, choose **Go to address** from the popup menu. RIDE displays a dialog box where you can type the address you want to reach.

4. **Search Symbol**: This command searches for the address of the selected symbol.

5. **Fill|Save**: This command permits the user to initialize a segment of the memory space with a constant number. It is possible to use this command to save the content of a segment to a file. Then this file could be used to initialize the segment again directly from the file.

6. **Toggle Read Breakpoint** : R This command will set a read breakpoint flag to the currently selected address. The program will be stopped when a program will try to read from this address.

7. **Toggle Write access Breakpoint**: W This command will set a write access breakpoint flag to the currently selected address. The program will be stopped when an instruction will try to write to this address.

8. If an address has both Read Access and Write Access flags, it is marked with H.

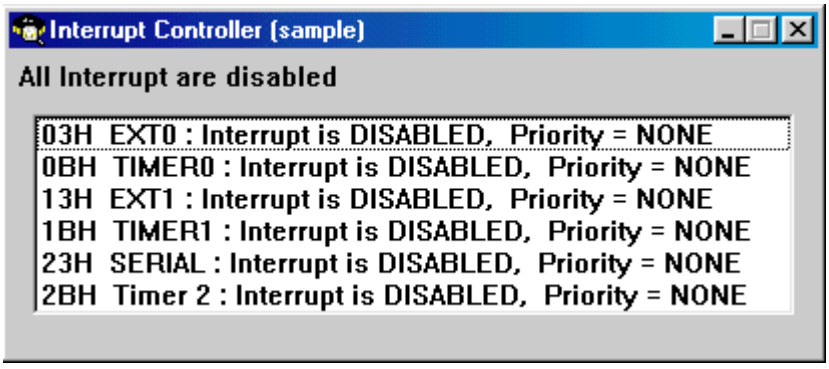
3.3.3 Hardware :

The **Debug|Hardware** command displays a submenu of the different peripherals of the selected microcontroller. Each command corresponds to a dynamic window.

There are 4 main categories of Hardware windows:

3.3.3.1 The interrupt controller:

The **Interrupt controller** window displays the current setting of each interrupt:

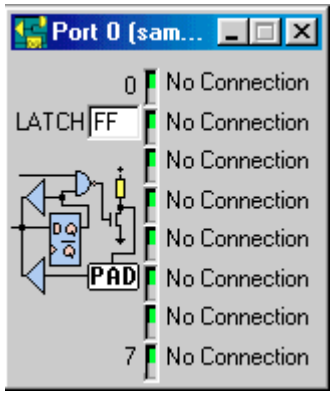


- Each interrupt is described on the following way:

Vector address Name: State, Priority level

3.3.3.2 I/O ports :

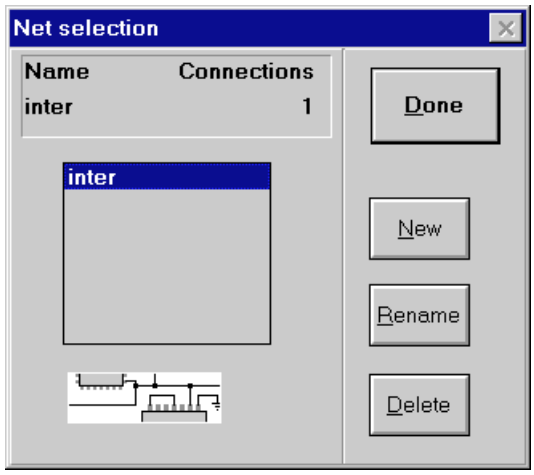
The port window displays:



- The current latch value in Hexadecimal (can be modified).
- The current value of each port is visualized with LEDs (Green = 1, Red = 0).

■ The **Link** command(select one of the 8 bits) allows to open a popup menu with 4 possibilities:

1. **No connection:** default value. No logic connection to **VCC**, **GND** or to a **NET**.
2. **Ground**: Logic connection to OV.
3. **Vcc:** Logic connection to 5V.
4. **Net:** Associates a logic connection: This command opens the following window:

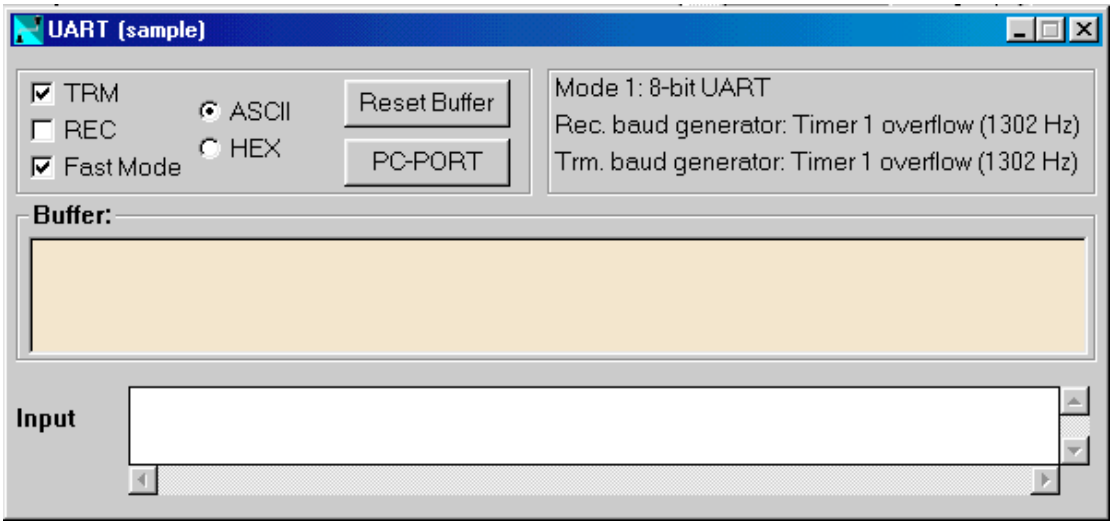


- Enter New to add a logic connection.
- Enter Rename to modify the name of the logic connection.
- Enter Delete to delete the logic connection.

In that way, you can « physically» link 2 ports with a logic connection.

3.3.3.3 The serial port(s) .

The **UART** window displays the following information:

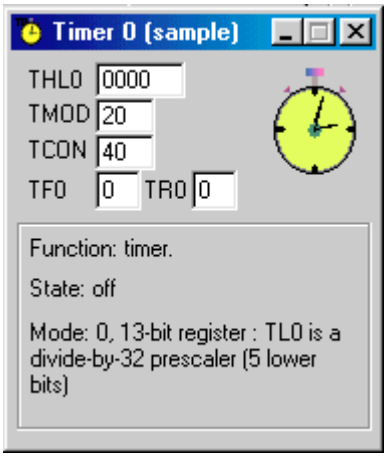


- The output buffer (« **Spy buffer ») ** shows in real time the character sent by the UART. The display can be **ASCII or HEX** (hexadecimal).
- **TRM** mode allows transmission (default value: on).
- **REC** mode allows reception (default value: on).
- The **Input Buffer** allows to simulate the reception of a character in the UART
  - In HEXA mode, Hexadecimal values can be separated with spaces.
  - In ASCII mode, the characters entered from the keyboard will be simulated in the input pin of the serial port.
  - The character \ is used as an Escape command for special characters as \n, \r and \t.
- **Fast Mode :** In that mode, the simulation of the serial port is made byte after byte (and not Bit after Bit).
- **Reset Buffer** erases the trace buffer.
- The right part describes the current UART setting.

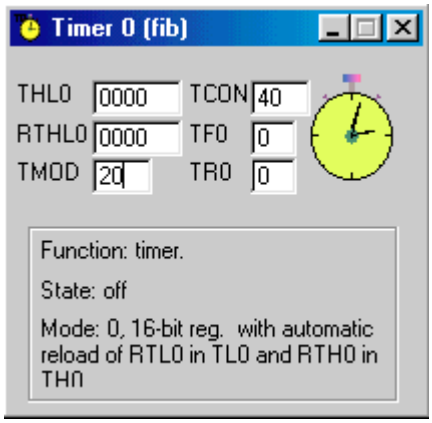
Note: The emitted and received characters will be simulated bit to bit in the I/O corresponding ports.

3.3.3.4 The timer(s):.

The Timer window displays the different registers associated to the timer and its current setting: Each register can be modified in this window.



8051



80C51XA

3.4 Other views

3.4.1 Map file

The **View|Map file** command displays the listing file (.M51or .MXA) corresponding to the project if it exists.

LX51 and RLXA generates listing files that contain the following information about the link process:

1. The invocation command line.
2. Object modules included in the project.
3. The memory map.
4. The overlay map.
5. The symbol table of public, local, and line number information.
6. A cross reference table.

3.4.2 Listing file

The **View|Listing** command displays the listing file (.**LST**) corresponding to the listing file generated for the currently selected node from the project or opened window.

3.4.3 View Stack

The **View|Stack** command displays the functions calling stack, during the execution of the program. This view is available for C functions.

Note: If the program includes nested functions, you have to compile with the OE2 option or pragma (extended debug information).

3.4.4 Report

The **View|Report** command displays various information depending on the status of the debugger.

3.4.5 Control flags list

This command displays a submenu where the user can select the type of flags he wants to view. There are three kinds of flags:

1. **Breakpoint**: where the program will stop.
2. **Trace**: where the trace information are recorded.
3. **Refresh**: where the debug views are refreshed.

Each view lists all the flags of the same type currently set in the loaded project.

1. The Flag views provides the following information about each flag:
2. Name of the source code file in which the flag is set.
3. Location (such as line number or address number) where the flag is set.
4. An eventual condition(available only for breakpoints).
5. The **enable** flag allows to enable or disable each flag from the list, during the debugging session.

The popup menu of the view gives access to the following commands:

1. **View source**: This command opens the source file where the flag has been set.
2. **Set Condition**: This command is available only for breakpoints. It displays a dialog box where the user can type the conditional expression to evaluate before pausing the program.
3. **Suppress**: This command deletes the currently selected flag.
4. **Delete All**: This command clears the list of flags.

4. Execution control

4.1 Running the program

Choose the command **Debug|Run** (short cut: Ctrl F9) to run the program from the execution point..

If you want to see the execution point while you are running the program, click the animating push-button in the tool bar. This command also refreshes in real time the opened debug windows (variables, registers, watches, etc...).

4.2 Stopping the program

Stopping the program means stopping the execution of the program. The program can be stopped with:

- a manual break (**stop** command)
- a breakpoint set in the source or in the code.
- a breakpoint with a verified condition
- a read breakpoint on a read access to a variable.
- a write breakpoint on a write access to a variable.

Stopping the program is symbolized with the execution point set on the source or code line where the break occurred.

The execution time will be stopped, and variables refreshed.

4.3 Stopping the execution

The execution is stopped with the command **Debug|Stop** (short cut Ctrl F9). The program is stopped at the end of the current line. If the current line is too long to execute, the **Break** button appears:

If you click this button, the execution is stopped inside the statement. The execution highlighted point will show you the next instruction to be executed inside the generated code.

Note: The different watched values won't be significant.

4.4 Running the program step by step

The step by step mode is divided in 2 modes:

1. **Step into**
2. **Step over**

4.4.1 step into mode

 The **Debug|Step into** command executes the program line by line. The execution point is displayed with a highlighted line in RIDE and shows you where you are in the execution of your program. The highlighted line corresponds to the next line to be executed.

If the execution of a line is too long the **Break** button becomes available in the tool bar. Clicking this button will stop the execution but the different watched values won't be significant.

4.4.2 Step over mode

 The **Debug|Step over** command executes the program line by line. The execution point is displayed with a highlighted line in RIDE and shows you where you are in the execution of your program. The highlighted line corresponds to the next line to be executed.

If the active window is a Code Window, the program will be executed code instruction by code instruction. This command doesn't run through the function calls.

4.5 Reset the program

 The **Debug|Reset** command (short cut Ctrl F2) resets the execution of the current program and initializes the program and data. The execution point will go to the beginning of the program.

4.6 Reset time

The command **Debug|Reset time** resets the simulation time displayed in the Message Bar of the desktop.

4.7 Control flags

A control flag is useful for the debugger to run predefined actions, at each time the program will execute the flagged line. The flag can be:

- **a breakpoint**: A « S » is displayed in the left side of the flagged line. Moreover, the line is red high-lightened. The program will stop just before executing this line.
- **a trace flag** : a « T » is displayed in the left side of the flagged line. Moreover, the line is green high-lightened. This flag is useful for the « Flashing » trace mode and the « On changes » trace mode. See chapter « Trace features ».
- **a refresh flag** : an electric light is displayed in the left side of the flagged line. At this flag, the opened debugging views will be refreshed.

☞ *Each flag can be enabled or disabled during the debugging session (**Enable** status in the Flags list view).*

4.8 Breakpoints

4.8.1 Program line breakpoint

The command **Debug|Add breakpoint** sets a breakpoint at the cursor location of the active Edit window. (Short cut: F5). The window can be the source or code window.

It is also possible to set a breakpoint with the popup menu.

The breakpoint is symbolized with a red highlighting upon the line to break.

When the program will stop on the line to break, the highlighting becomes pink.

To remove a breakpoint: Do the same way (same command, short cut and button).

To view the breakpoints list: Press the **View|Flags|Breakpoints** command.

4.8.2 Variable access breakpoint

To set a breakpoint on the access to a variable located in the following spaces: XDATA, DATA, SFR or BIT

- open the memory space including the variable (with View|Data dump command).
- open the popup menu and choose **search symbol**
- When your cursor is on the variable choose in the popup menu:

1. **Toggle flag / Toggle read breakpoint:** This command will set a read breakpoint flag to the currently selected address. The program will be stopped when a program will try to read from this address.

2. **Toggle flag / Toggle write breakpoint:** This command will set a write access breakpoint flag to the currently selected address. The program will be stopped when an instruction will try to write to this address.

4.8.3 Conditional and complex breakpoint

This command is available only for breakpoints. It displays a dialog box where the user can type the conditional expression to evaluate before pausing the program. To set a condition:

- open the Breakpoint list window, with the **View|Flags|Breakpoints** command.
- select the breakpoint from the displayed list.
- open the popup menu (mouse right button) and choose the **Set condition** command.
- enter the condition in the input dialog box: the syntax is the same as C language syntax (ex: var == 1).

5. Trace features

5.1 Introduction

5.2 Invocation

5.3 Trace options

5.1 Introduction

When you debug a program, you sometimes need to know what happened « before », to explain the current program state. The trace features allow to load in memory the execution of the program. After a breakpoint, you can display the trace buffer that will show the executed lines, and some others information, about the time, or the value of variables.

5.2 Invocation

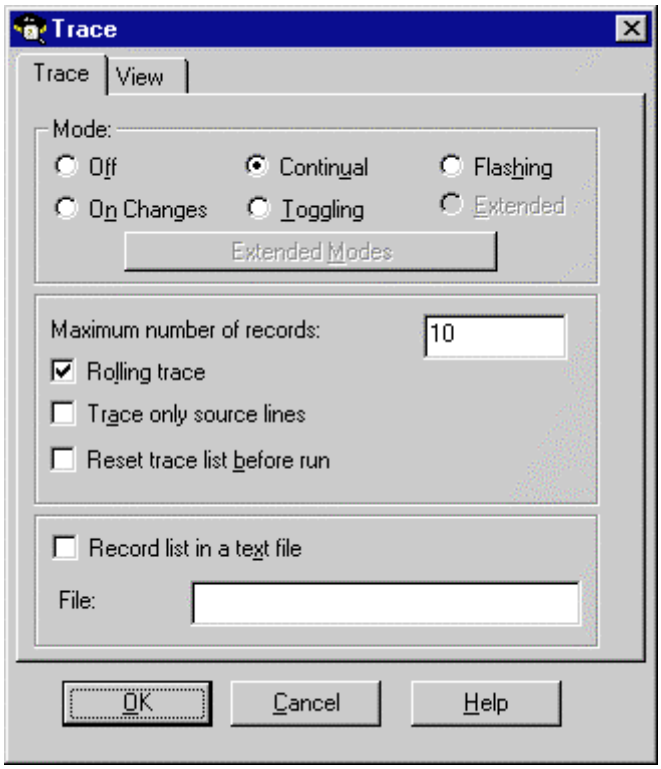
2 trace modes are available:

- **Continual:** This mode corresponds to a continual recording of the trace information independently of trace flags.
- **Triggered with a trace flag:** The trace flag gives the start, or stop for recording into the trace buffer. Refer to the next chapter to choose the trace option.

The **Debug|Toggle trace** (short-cut F4) command inserts a trace flag to the currently selected source or code instruction in the active Edit Window or Code Window. If the currently selected line doesn't correspond to a valid instruction the flag will be set to the next valid address. The corresponding line is marked by **T**.

5.3 Trace options

The command **Options|Trace** is available during the debug session. This command displays the dialog box « Trace|Options » where the user can set the different parameters for the trace feature. It is possible to display this dialog box by clicking the right button of the mouse in the records list area of the Trace window.



5.3.1 Mode:

Off: No trace recording.

Continual: This mode corresponds to a continual recording of the trace information independently of trace flags.

Flashing: Under this mode RIDE will record the trace information each time the program executes a line marked with a trace flag.

Toggling: This mode corresponds to a toggling mode of recording. The first time a trace flag is met during the execution of the program, RIDE toggles the trace mode to On and begins recording the trace information instruction by instruction until meeting another trace flag. Then the trace mode is toggled to Off and so on.

On Changes: Under this mode of tracing, RIDE will record the trace information only if one of the selected watch expressions has been modified by the current instructions.

5.3.2 Trace view

The **View|Trace** command opens the Trace window. The window is divided in 2 parts: A list of track records, and a graphical trace.

The list of trace records:

This list contains different columns:

- Time
- The spent between the currently selected records and the next one.
- PC.
- The source line.

The value of the selected watched expressions.

The width of each part could be modified by scrolling the corresponding bars in the title.

The graphical trace

The size of the graphical part could be modified with scrolling the separator bar.

To trace an expression click the corresponding button in the title bar of the trace records. To suppress a graphic click the button again. The graphical trace is available only for simple typed expressions. The expressions which have a graphical representation are marked with  .

When the program is stopped, it is possible to zoom a region of the graphical trace by dragging with the mouse a rectangle inside of this zone. To return to the initial position click the right button of the mouse.

5.3.3 Other options

maximum number of recordings:

The number of trace recordings is limited. This limit is set by default to 10 but could be modified by the user up to 9999. The higher the number, the slower the manipulation will be.

Rolling trace:

If this option is not selected RIDE will stop recording when the number of records reaches the preceding limit. If the option is selected, it will continue to record by suppressing the oldest record.

Display the number of cycles:

If this option is selected the time will be displayed as the number of machine cycles.

Register only lines:

If the trace mode is set to **Continual** or **Toggling** with this option only the instructions corresponding to a source line will be recorded.

Display the disassembly code:

If this option is selected the disassembly code of the source line will be displayed.

Display the relative time:

If this option is selected the time will be displayed relative to the selected line.

The **Reset** button will flush the entire record list.

The **Ports** button is available for the emulation and will let the user select the ports bit to be displayed.

Reset before Run

This option will automatically reset the trace list before running again the program again.

6. Bank switching

6.1 Bank switching with RIDE IDE

6.2 Debugging an application using bank switching

6.1 Bank switching with RIDE

The RKitE51 (Enterprise kit) is supplied with the LX-51 Banking linker, which can generate bank switching code.

The linker (with the **Build All**, **Make** or **Link** command) generates a file with suffix AOF for the basic space, and as many files with the suffixes Xk(.X1,...) as there are open extended spaces. The figure k corresponds to the reference of the open space.

Moreover, RIDE will generate a LIS file, necessary to debug the bank switching application with the Simulator and the emulator.

6.2 Debugging an application using bank switching

In the debug options, 2 important configurations must be set:

- The **Code size** : put always 64 KB.
- In the Emulator options, click on MODE32 for a 32KB pages size mode. The default value is 64KB.

In the debugger, the **XAREA** variable, in the Main registers window, shows the current page number.

In the Code window, the address specifies the page number (ex 01:8000 for address 8000 of page 1).

In simulation, switch flags (« B ») are set automatically in front of the source line where the page is changing.

7. Options summary

7.1 Debugging options

7.2 TRACE options

7.3 Peripheral options

7.1 Debugging options

The **Debug|Options** opens the **Debugger Options** dialog box, where you can choose the debug tool and set the tool parameters.

In order to set the debugger options, you can use the following process:

1. If you want to use the simulator, select **Virtual Machine(Simulator)**
2. If you want to use the emulator, select **Real Machine**
3. If you want to use another debugging tool choose **Other Tool.**

For more information, see chapter 2, section 4.

7.2 TRACE options

See chapter 5, section 3.

7.3 Peripheral options

This command is available only under the debugging mode where the selected debugger tool is the Virtual machine (simulator). This command will display the **Peripheral** : dialog box where the user can select the peripherals actually simulated.

The simulated peripherals correspond to the highlighted lines. To deselect or select a peripheral click the corresponding line. These selections will be lost from a debug session to another.

This option is very useful to accelerate the simulation speed, when the program doesn’t use all the peripherals.

8. Commands summary

COMMANDS	SHORT CUTS	BUTTON	DESCRIPTION
Options commands			
Start	Alt D S		Starts the debugging mode
Animate			Animates the execution of the program
Refresh	Alt W R		Refreshes the debug windows
Reset	Alt D E or **Ctrl F2**		Resets the program
Run	Alt D R or **Ctrl F9**		Runs the program
Stop	Alt D S or **Ctrl F9**		Stops the program
Step into	Alt D I or **F7**		Executes the current statement
Step over	Alt D O or **F8**		Executes the current statement
Break			Unconditional break on the program
Add watch	Alt D A or **F6**		Add a new watch expression
Toggle/ remove Breakpoint	Alt D K or **F5**		Add/ remove a new breakpoint
Toggle/ remove Trace	Alt D G or **F4**		Add/ remove a new Trace flag

9. Index