ARM

series

SIGNUM SYSTEMS CORPORATION

Using JTAGjet-ARM with Keil µVision

Installation Instructions

SIGNUM SYSTEMS

COPYRIGHT NOTICE

DISCLAIMER

Signum Systems makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Also, Signum Systems reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Signum Systems to notify any person or organization of such revision or changes.

WARRANTY

Signum Systems warrants to the original purchaser that this product is free of defects in material and workmanship and performs to applicable published Signum Systems specifications for a period of SIX MONTHS from the date of shipment. If defective, the product must be returned to Signum Systems, prepaid, within the warranty period, and it will be repaired or replaced (at our option) at no charge. Equipment or parts which have been subject to misuse, abuse, alteration, neglect, accident, unauthorized installation or repair are not covered by warranty. This warranty is in lieu of any other warranty expressed or implied. IN NO EVENT SHALL SIGNUM SYSTEMS BE LIABLE FOR CONSEQUENTIAL DAMAGES OF ANY KIND. It is up to the purchaser to determine the reliability and suitability of this product for his particular application.

SIGNUM SYSTEMS
11992 CHALLENGER COURT
MOORPARK, CA 93021, U.S.A
PHONE 805 • 523 • 9774
WWW.SIGNUM.COM

Table of Contents

Driver Installation

1. Insert the Signum *JTAGjet* CD into the CD-ROM drive. In the Master Setup dialog box, select JTAGjet Drivers for Third-Party Debuggers > Keil µVision3/µVision4 Driver for ARM (Figure 1).

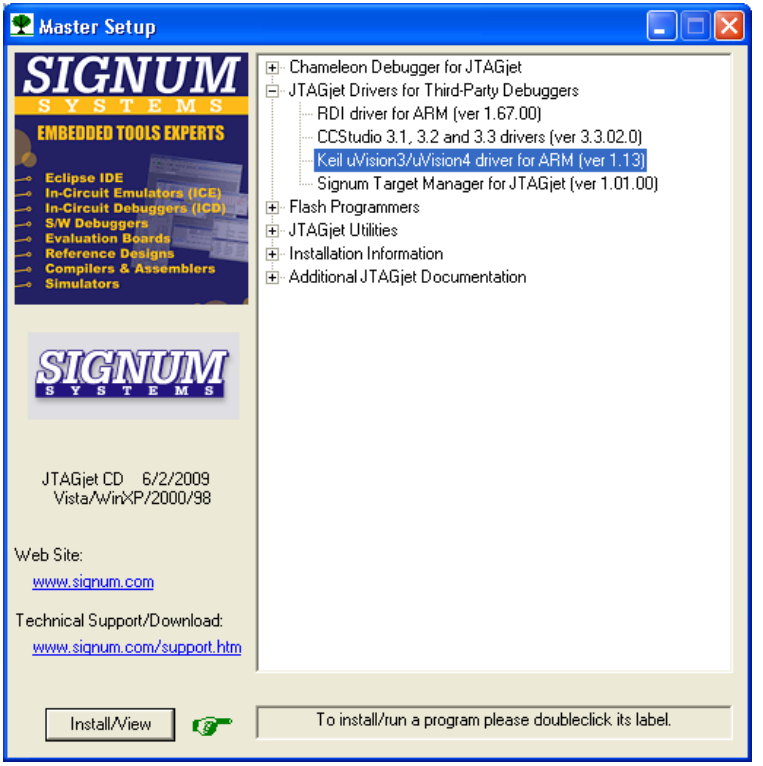


FIGURE 1 Master Setup dialog box.

2. In the initial "Welcome to Keil µVision3/µVision4" dialog box, click the Next button to start the installation process.

3. Accept the End-User Software License Agreement. Click Next.

4. The driver should be installed where µVision3/µVision4 is installed (Figure 2). Choose the appropriate destination folder and click Next.

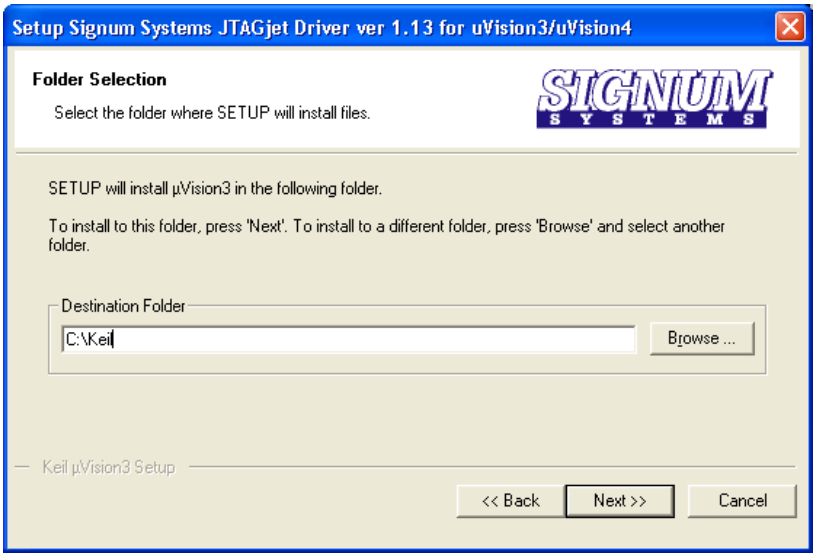
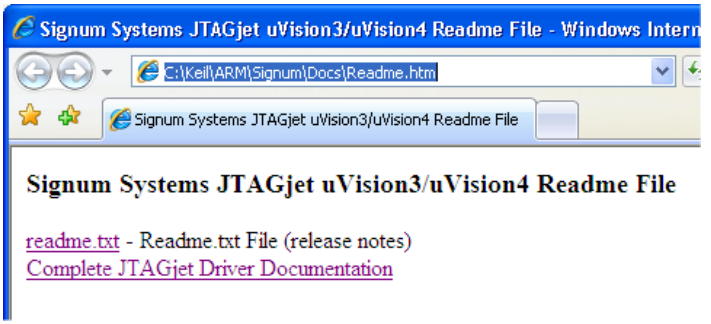


FIGURE 2 Selecting the driver installation folder.

5. In the Customer Information dialog box, verify the displayed information and click Next.

6. Installation starts. At the end of the process, the Keil µVision3/µVision4 Setup Completed dialog box appears. The Readme file contains a list of the changes in the current release code and a complete driver documentation.



The complete JTAGjet Driver documentation is also accessible in µVision in the Help > Open Books window.

7. Click Finish. The JTAGjet ARM Driver for Keil µVision3/µVision4 package is installed.

8. Install the JTAGjet USB driver. Connect the emulator to computer's USB port. when prompted for the driver location, navigate to the CD root folder or C:\Keil\ARM\Signum\USB Driver subfolder of the main µVision3 installation folder. The required SigUSB.inf and corresponding *.sys files are located there. For more details, see *USB 2.0 Driver for JTAGjet and ADM51: Installation Instructions*.

9. The Keil µVision3 Driver for ARM requires license permissions for the emulator. Some emulator models have the license built into them, eliminating thus the need for the user's intervention. However, emulators without integrated license support display the error message shown in Figure 3 upon µVision3's attempt to establish connection with them.

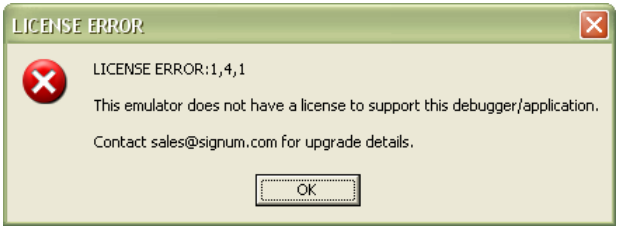


FIGURE 3 License error message.

The license can be purchased either from your local Signum distributor or directly from Signum Systems Corp. For a world-wide list of distributors, visit www.signum.com/distribu.htm.

The .lic licenses files are tied to a specific emulator serial number (embedded in the filename). Before ordering, please verify the serial number on the bottom of your emulator. Once obtained, the file should be placed in the driver installation directory, i.e., where SigUV3Arm.dll resides. The default directory is C:\Keil\ARM\Signum.

Working with Existing µVision3 Projects

Keil's comes with several sample applications for popular ARM evaluation boards manufactured by Keil and other vendors. This brief tutorial demonstrates how to use the JTAGjet-ARM emulator with the MCB2100 Evaluation Board from Keil. The target is equipped with the Philips LPC2129 ARM7 microcontroller.

Using JTAGjet, Blinky RAM & MCB2100

Start by opening the Keil µVision3 IDE. Open the Blinky RAM example compiled with the Keil C compiler by selecting Project > Open Project. Navigate to the Blinky.uv2 file located in the ARM/Boards/Keil/MCB2100/Blinky subfolder of the main µVision3 installation folder. Make sure that MCB2100 RAM configuration is selected in the Select Target combo box on the IDE's toolbar. The Project Workspace window lists the project files (Figure 4).

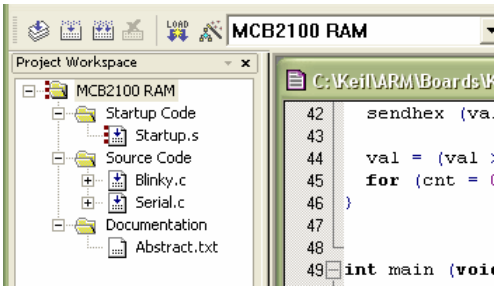


FIGURE 4 The MCB2100 Blinky RAM application.

In the Project Workspace window, right-click the MCB2100 RAM item and select Options for Target MCB2100 RAM menu item. This opens Options for Target 'MCB2100 RAM' dialog box. It allows you to select the following tabs:

- **Device** — controls the target processor for which code should be generated.
- **Target** — controls the target memory layout.
- **Output** — selects the name and format of the output application.
- **Listing** — selects types of listing files generated when building the project.
- **C** — contains the compiler options for translating project files written in C.
- **Asm** — provides assembly language options for translating the startup project.
- **LA Locate** — allows you to choose the final application placement in target memory.
- **LA Misc** — provides access to various linker and locator options for linking the output file.
- **Debug** — allows to configure the debugger.
- **Utilities** — provides access to the flasher used to write an application to target flash memory. Since Blinky RAM application is configured to be loaded only into the LPC2129 RAM, these settings are of no interest to us.

The settings provided for this project are designed to let you use the Blinky application with the JTAGjet emulator immediately. However, it is necessary to select the debug interface. To do that, click the Debug tab. Select „Use Signum

Systems JTAGjet Driver" from the list of available debug interfaces (the fields highlighted in red in Figure 5). The "Load Application on Startup" and "Run to main()" check boxes should remain unchecked. The "Initialization File" field should be set to RAM.ini (see the fields highlighted in green in Figure 5).

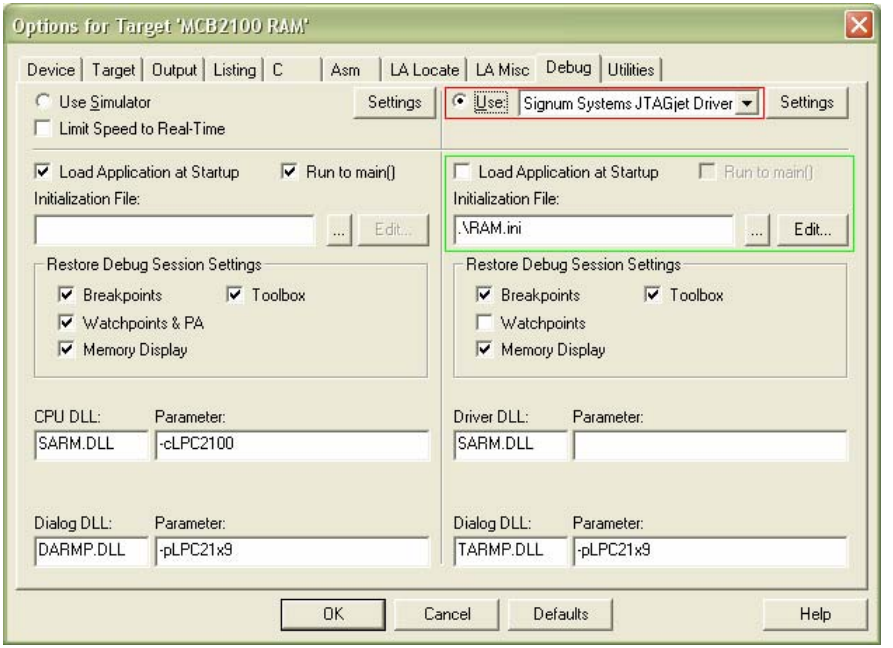


FIGURE 5 Selecting the Signum Systems JTAGjet Driver for debugging the MCB2100 Blinky RAM application.

As a result, when the debugger connects to the target, µVision3 executes commands in the file RAM.ini, but does not load the application into the target memory. The code is not executed until the main( ) function.

The explanation of why the "Load Application on Startup" and "Run to main()" check boxes remain unchecked is found in the RAM.ini file. Select Edit to take a closer look at its contents. The file contains commands that load the application into the target memory, set the PC at the code starting point (address 0x4000 0000) and then execute the startup code from its starting point till the entry point of the main( ) function, as shown below.

```
FUNC void Setup (void) {
```

```
// <o> Program Entry Point
  PC = 0x40000000;
}

LOAD RAM\Blinky INCREMENTAL          // Download

Setup();
// Setup for Running

g, main
```

We recommend keeping the other original project settings unchanged when building the Blinky RAM application. If you want to experiment with these settings, try code optimization levels accessible through the C tab. The default setting is "Level 7 – loop strength reduction." Applying the lowest optimization, "Level 0 – constant folding," results in larger code. However, it will allow you correlate the ARM instructions with individual statements in C after the application is loaded into the target memory.

You can build the entire project by selecting Project > "Rebuild all target files" from the µVision3 menu. To start the µVision3 debugger with the newly created "Blinky RAM" application, select Debug > Start/Stop Debug Session from the µVision3 menu or click the "Start/Stop Debug Session" button. A sample debug session is described in detail in chapter *Sample Debugger Session* on page 31.

Using JTAGjet, Blinky FLASH & MCB2100

Open the Blinky Flash example compiled with the ARM RealView C compiler by selecting Project > Open Project. Navigate to the file Blinky.uv2 located in the ARM/RV30/Boards/Keil/MCB2100/Blinky subfolder of the main µVision3 installation folder. Make sure that the MCB2100 Flash configuration is selected in the Select Target combo box on the IDE's toolbar. The Project Workspace window lists the project files (Figure 6).

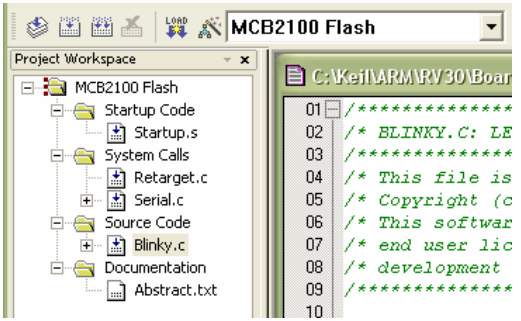


FIGURE 6 µVision3 Project Workspace window showing OPENED MCB2100 Blinky Flash application.

In the Project Workspace window, right-click the MCB2100 Flash item and select "Options for Target MCB2100 Flash." This opens the Options for Target 'MCB2100 Flash' dialog box containing nine tabs. (This example is compiled using the ARM RealView tools, whereas the previous one was compiled with the Keil C compiler). These tabs are as follows:

- **Device** — selects the target processor for which code is generated.
- **Target** — controls the target memory layout.
- **Output** — allows you to select the name and format of output application.
- **Listing** — allows you to select the types of listing files to be generated when building the project.
- **C/C++** — allows you to modify the compiler options for translating C/C++ language project files.
- **Asm** — Assembly language options for translating the startup project file.
- **Linker** — controls the placement of the application in the target memory.
- **Debug** — controls the debugger configuration.
- **Utilities** — controls the flasher for writing the application to the target flash memory.

Since this example is compiled using a compiler different from that in the previous example, the contents of the dialog box tabs differs, accordingly. Most settings provided with this project are appropriate for the Blinky Flash

application to run with the JTAGjet emulator. The only setting that requires modification is the selection of the debug interface and the flasher.

To select the target debug interface, click the Debug tab of the Options for Target 'MCB2100 Flash' dialog box and select "Use Signum Systems JTAGjet Driver" from the list of available debug interfaces (the fields highlighted in red in Figure 7). Note that now the "Load Application on Startup" and "Run to main()" check boxes are selected. In addition, the "Initialization File" field is left empty. (Green highlight in Figure 7.)

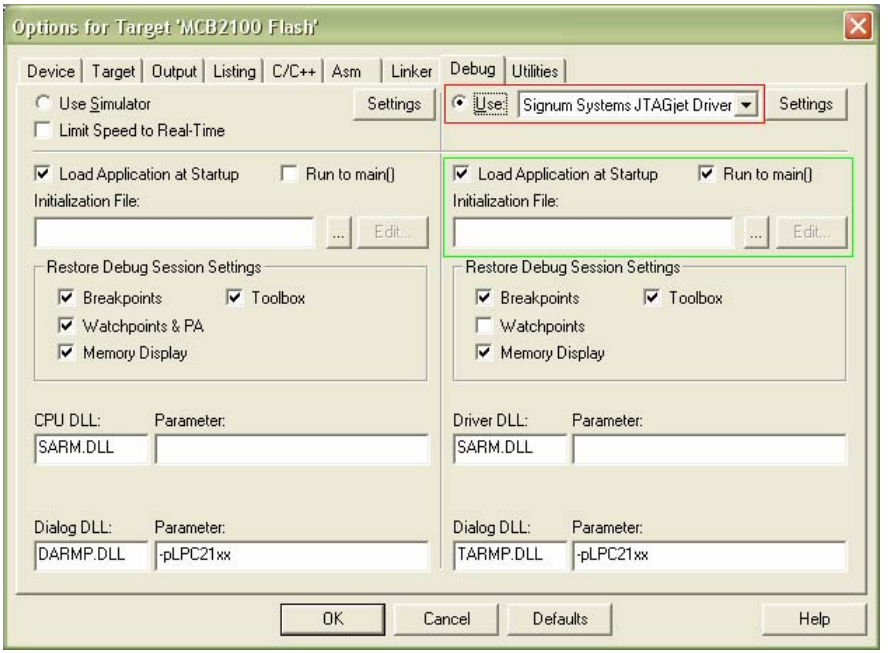


FIGURE 7 Selecting Signum Systems JTAGjet DRIVER for debugging MCB2100 Blinky Flash application.

As a result, µVision3 will load the application debug information when debugger connects to target, and then execute the startup code from its starting point till the entry point of the main( ) function. (The IDE will not write the code into the Flash memory, however. This will be accomplished in a separate step.)

The ARM7 default starting address is 0x0. Since the Flash starts at this address and the startup code was also compiled to have starting point at 0x0, no additional step is required to set up the initial PC value.

To select the target flasher application, in the Options for Target 'MCB2100 Flash' dialog box, click the Utilities tab. Select the "Use Target Driver for Flash Programming" and "Signum Systems JTAGjet Driver" from the list of available drivers (red highlight in FIGURE 8).

Note that in the original project settings, the Update Target before Debugging check box is selected and the Init File field empty (green highlight in Figure 8). Consequently, the target flash memory is programmed with the latest application binary code every time the debugger connects to target. No special commands are executed.

To avoid programming target flash memory when the debug session starts, uncheck the "Update Target before Debugging" check box in this tab. Then load the application code to the Flash manually when required using the IDE's Flash menu. This method may prove useful when the application does not change, or the number of safe Flash reprogramming cycles is limited.

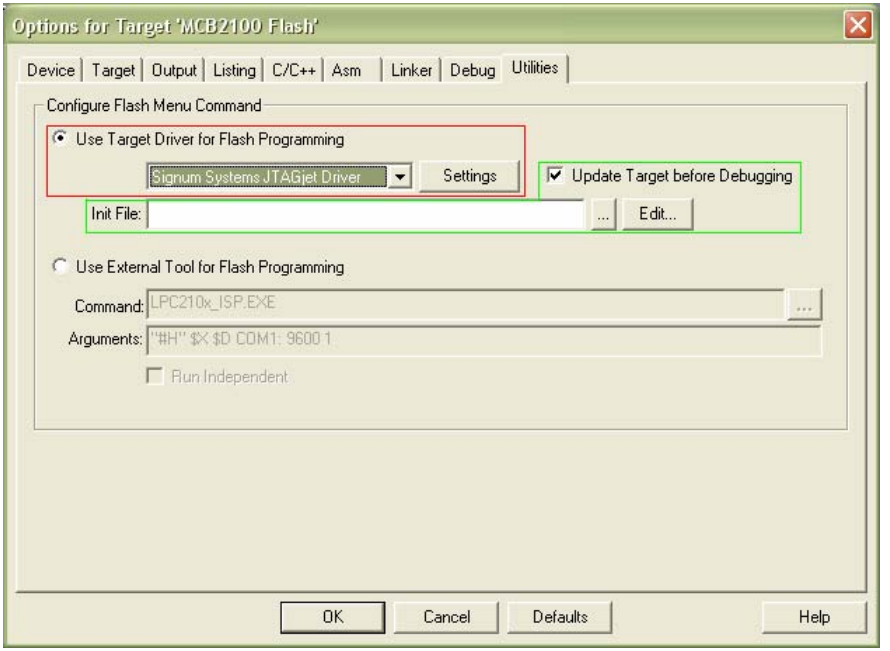


FIGURE 8 Selecting the Signum JTAGjet Driver as the flash programmer for the MCB2100 'Blinky Flash' project.

Press the **Settings** button (red highlight in Figure 8) to configure the flasher. See *Flash Programming* on page 21 for details.

Changing any other original project settings for the 'Blinky Flash' application with RealView tools is not recommended. To experiment with these settings, try changing the code optimization level selectable from the C/C++ tab. The default is Level 3. Level 0 produces larger code that allows you to correlate the ARM instructions generated for specific C language statements after loading the application into the target memory.

Build the entire project by selecting Project > Rebuild All Target Files from the µVision3 menu. The 'Blinky Flash' application is built and ready to be burned into the Flash. The Flash programming process is described in detail in chapter *Flash Programming* on page 21.

To start the µVision3 debugger with the newly compiled 'Blinky Flash' application, Start/Stop Debug Session from the Debug menu. Alternately, click the "Start/Stop Debug Session" button on the µVision3 toolbar. For a sample debug session, see Sample Debugger Session on page 31.

JTAGjet Debug Projects

The preceding sections explained how to run and adapt the two existing µVision3 sample applications for the JTAGjet emulator.

In this section, we will create two application programs from scratch. The first of them will be executed in the RAM of the LPC2129 ARM7 processor. The other one will run in the Flash of the same microcontroller.

"Hello RAM" Application for MCB2100

Let us create a simple HelloR program in C that places the string "Hello World" in a global text buffer variable. The application will be built with the RealView compiler to be executed in the RAM of the LPC2129 ARM7 processor.

The step of the process is to create a new project under Keil µVision3 IDE. Since HelloR is to be compiled with RealView, make sure that the RealView

tools are selected as current development tools when the project is created. Keil µVision3 IDE supports ARM RealView, Keil CARM tools and GNU tools. To verify which of them are in force, select Project > Components, Environments, Books. The Components, Environment and Books dialog box appears. Verify that RealView tools are indeed selected (Figure 9).

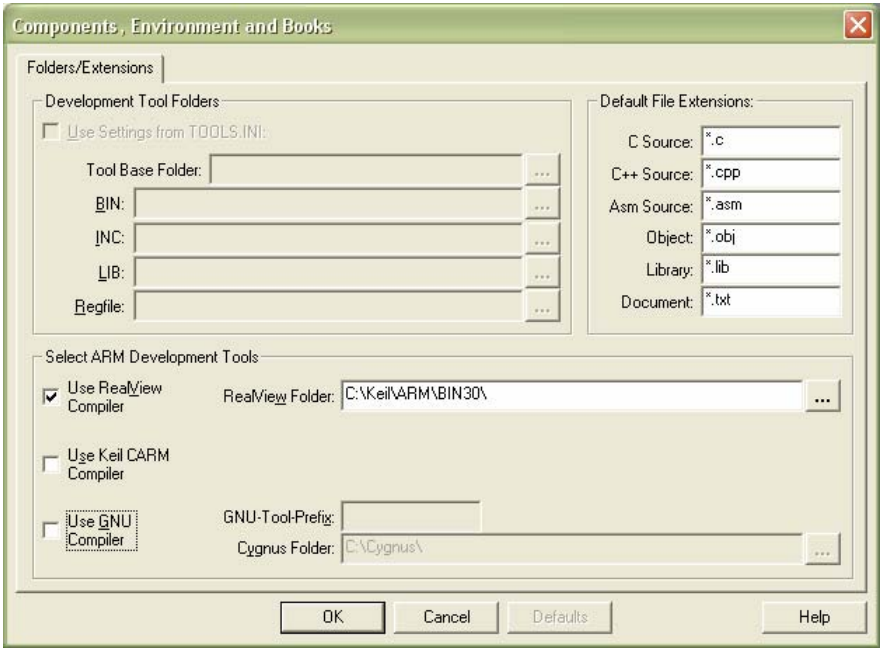


FIGURE 9 Verifying that the ARM RealView compiler is the current ARM compiler for µVision3.

To create the new project, select New Project from the Project menu. Navigate to the folder assigned to the project (C:\Keil\ARM\RV30\Examples\JTAGjet\HelloR) and select HelloR as the new project name (Figure 10).

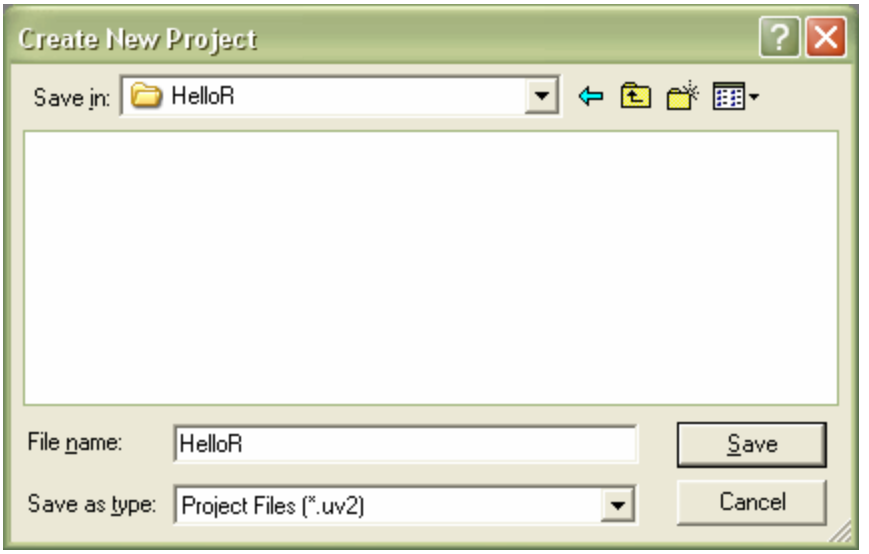


FIGURE 10 Creating the new project named HelloR.

After the new project has been created, µVision3 prompts the user to select target processor. In the displayed list, find Philips and select the LPC2129. Click OK. Respond Yes to the next prompt to include the startup file in the project automatically by copying the Philips LPC2100 startup code to the project folder (Figure 11).

FIGURE 11 Adding automatically STARTUP file when creating new project.

The new project HelloR with one default project target has been created. The project target is named Target1 by default. It appears in the Project Workspace window as well as in the Select Target combo box located on the IDE toolbar. To rename the project, click its name and type over it "Hello RAM." The new name appears on the IDE toolbar (Figure 12).

FIGURE 12 The µVision3 PROJECT Workspace window displays the newly created Hello RAM target of the HelloR project.

Let us add to the project a new C file with the main() function in it. Select File > New and type the following text in the new Text1 window using µVision3 editor:

```
#include <string.h>
char buff[256] = { 0 };

int main (void)
{
    int i;
    for (i = 0; i < sizeof(buff); i++) {
        buff[i] = '\0';
    }
    strcpy( &buff[0], "Hello World" );
    while (1) {
        i++;
    }
}
```

This simple code clears the buff[ ] buffer, copies the string "Hello World" into it, and enters an infinite loop that increments the local variable "i".

Save the window contents into the main.c file in the HelloR project folder by selecting File > Save As. To add the newly created file to the project:

- Right-click "Source Group 1" item in the Project Workspace window. A pop-up window appears.
- Select Add Files to Group 'Source Group 1.' The Add Files to Group 'Source Group 1' dialog box appears.
- Select "main.c" and click Add (Figure 13).

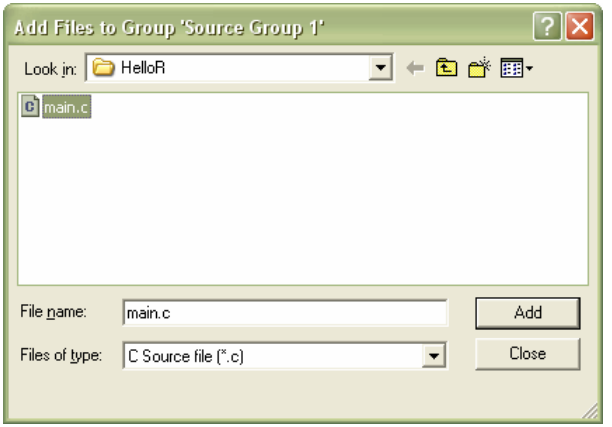


FIGURE 13 Adding the newly created main.c file to the project.

The Project Workspace window displays the two files, startup.s and main.c, for the "Hello RAM" target of the HelloR project (Figure 14).

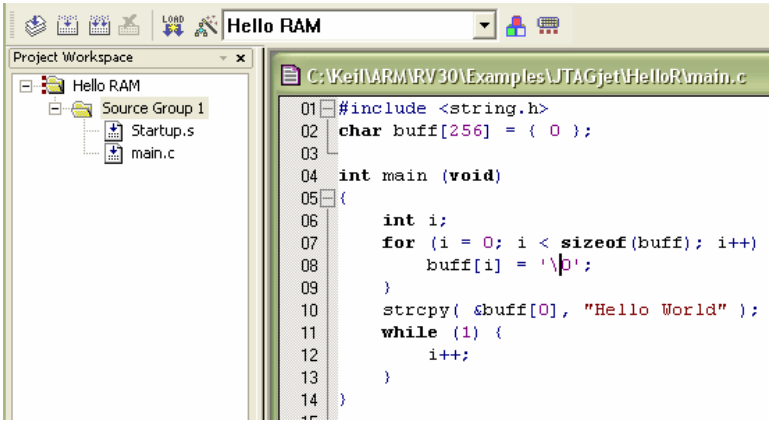


FIGURE 14 The Hello RAM target of the HelloR project consists of two files: startup.s and main.c.

With the project code ready, let choose the project compilation and linking options and build the application. To execute in RAM, the application must be linked at the RAM starting address 0x40000000.

Right-click the Hello RAM item and select "Options for Target 'Hello RAM'." The Options for Target dialog box opens. Select the Linker tab. As both the

code and the data must be loaded to RAM, set the R/O Base to 0x40000000 (the RAM start address). Similarly, set the R/W Base to 0x40003000 (address in the upper part of the RAM). Do not modify the other settings on the Target, Output, Listing, C/C++ and Asm tabs (Figure 15).

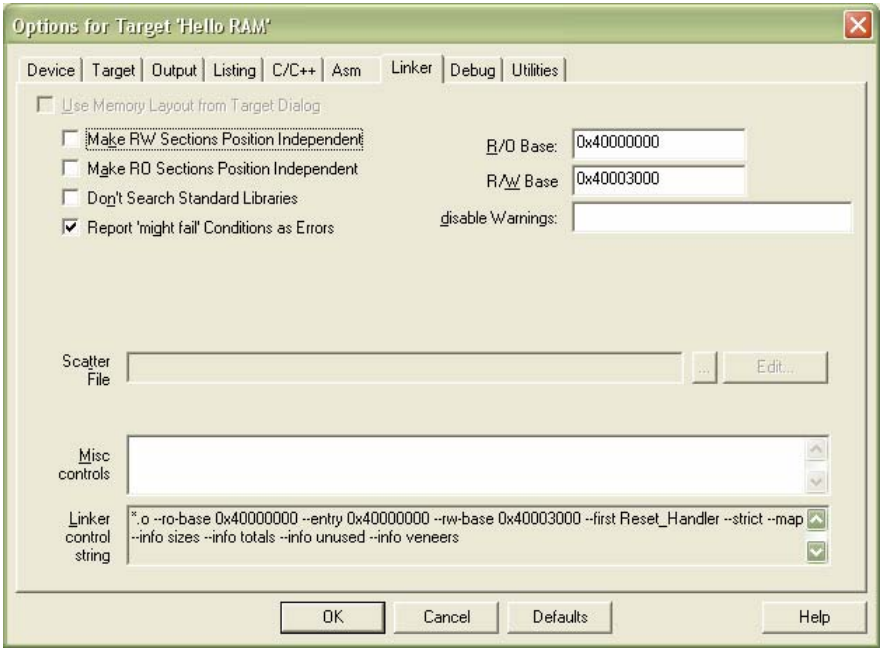


FIGURE 15 Setting the starting addresses for RAM-based application (R/O Base) and for the data sections (R/W Base).

While in the Options for Target dialog box, let us choose the correct debugger. In the Debug tab, select Use Signum Systems JTAGjet Driver from the list of available debug interfaces (Figure 5). Leave the "Load Application on Startup" and "Run to main()" check boxes unselected. In the Initialization File edit box, type "\RAM.ini".

We will create a command file executed when the debugger connects to the target. In a Windows Explorer window, navigate to the HelloR project folder and create an empty text file named RAM.ini. Back in the µVision3's "Options for Target" dialog box, click the Edit button located at right hand side of Initialization File edit box. The initially empty file RAM.ini will be opened for

editing in µVision3 client area. Close the dialog box and type the following commands into the editor window with RAM.ini in it:

```
FUNC void Setup (void) {
// <o> Program Entry Point
  PC = 0x40000000;
}

LOAD HelloR.axf INCREMENTAL      // Download

Setup();                         // Setup for Running

g ,main
```

After the debugger connects to the target, these commands will load the application file HelloR.axf into the target RAM memory, set the initial value of the PC to the RAM beginning address 0x40000000, and execute the program till the main( ) routine.

Build whole project target (HelloR) by selecting Project > Rebuild all Target Files menu item. Section *Sample Debugger Session* on page 31 explains how to debug the resulting HelloR program.

"Hello FLASH" application for MCB2100

Let us modify the HelloR project so that instead of running in RAM, the code is executed in the internal Flash memory of the LPC2129. We will create a second project target, Hello Flash, to accomplish this.

From µVision3 Project menu, open the HelloR project and then select Project > "Components, Environment, Books." This will open the "Components, Environment and Books" dialog box. Click the Project Components tab. The tab displays in columns the existing project targets (currently, "Hello RAM"), the file groups associated with the selected targets ("Source Group 1"), and the files belonging to the selected file groups (Startup.s and main.c).

To add a new project target, click New (Insert) button located at right hand side of text "Project Targets." Name the target "Hello Flash" (Figure 16). By default, newly created project target shares the groups and source files with existing target Hello RAM. This makes the new target ready for use almost instantly.

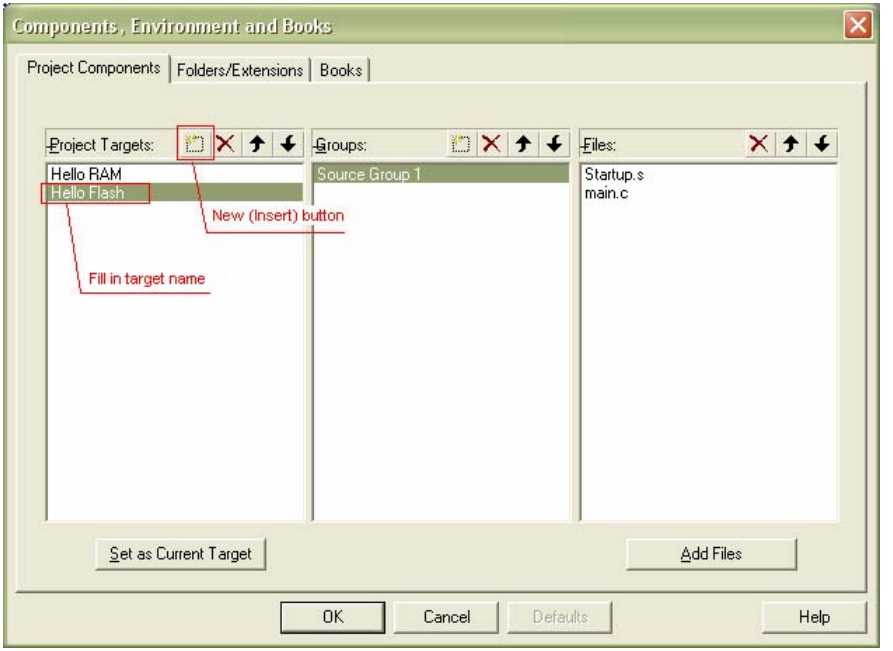


FIGURE 16 Adding the new project target "Hello Flash" to an existing project.

Close the dialog box. In the Select Target combo box on the µVision3 toolbar, select the Hello Flash. The current target displayed in the Project Workspace window changes accordingly (Figure 17).

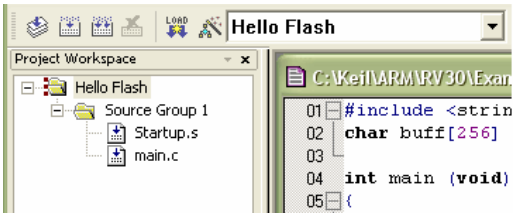


FIGURE 17 The "Hello Flash" project target shares files with the "Hello RAM" project.

We will now configure the building, debugging and flash programming settings. In the Project Workspace window, right-click Hello Flash. From the pop-up window, select "Options for Target 'Hello Flash'." The "Options for Target" dialog box opens. Click the Output tab. Name the executable HelloF to keep the HelloR application intact when building its flash target.

Click the Linker tab. Set R/O Base to 0x0 to put the code at the start of the Flash. Set R/W Base to 0x40000000 to load the data where the RAM begins. Accept without any modifications other options on the Target, Listing, C/C++ and Asm tabs (Figure 18).

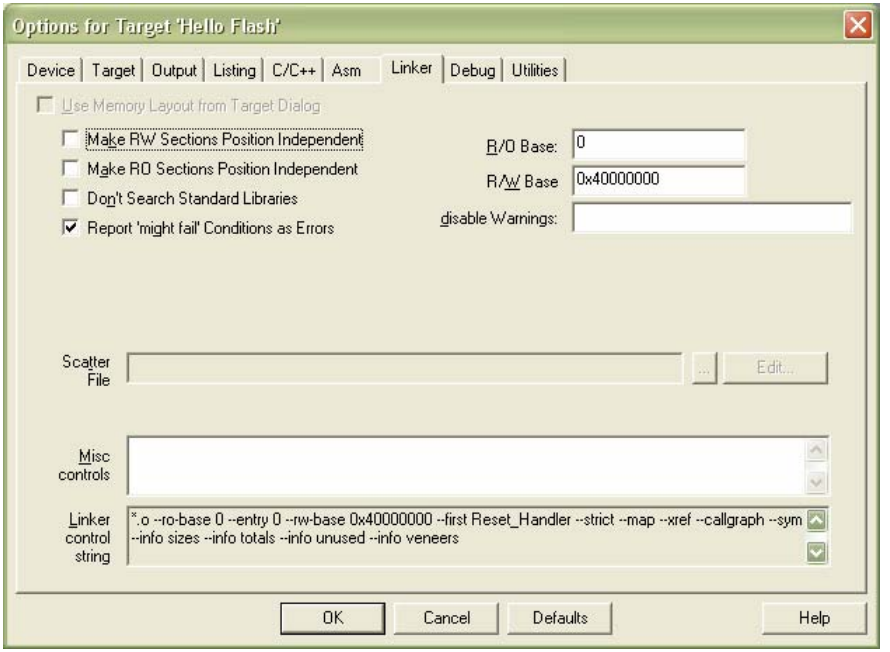


FIGURE 18 Setting the starting addresses for the flash-based application code (R/O Base) and data (R/W Base).

Select now the debugger and flasher.

Click the Debug tab and select "Use Signum Systems JTAGjet Driver" from the list of available debug interfaces (Figure 5). Select both "Load Application on Startup" and "Run to main()" check boxes. Make sure that the Initialization File edit box is left empty to make µVision3 load the application debug information at the beginning of the debug session and execute the application code address 0 till the main ( ) function.

Click the Utilities tab to select the flasher for writing HelloF the LPC2129 on-chip flash memory. Click the "Use Target Driver for Flash Programming" radio button, and then select "Signum Systems JTAGjet Driver" from the target driver

list. Leave the "Update Target before Debugging "check box unchecked (Figure 19).

Press the OK button to accept the settings and close dialog box.

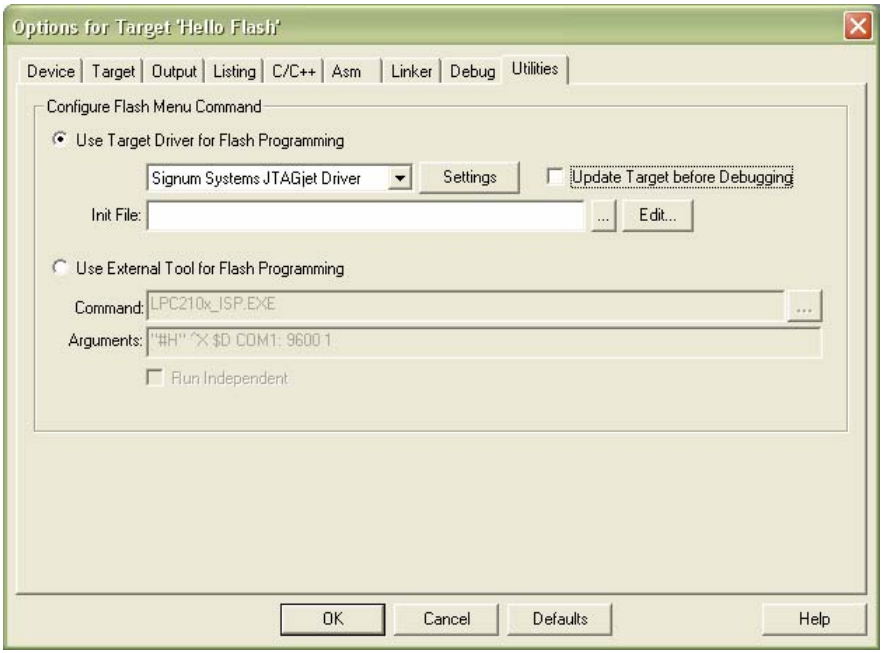


FIGURE 19 Selecting the Signum JTAGjet Driver as the flasher for the Hello Flash target of the HelloR project.

Build the project by selecting Project > "Rebuild all target files" from the µVision3 menu. The Hello Flash application is created and can be burnt into the Flash. For detail, see Flash Programming on page 21. When the flash memory is programmed with the Hello Flash application, start a debug session by selecting "Debug > Start/Stop Debug Session" from µVision3 menu. Alternately, click the "Start/Stop Debug Session" button on the µVision3 toolbar. For a sample debug session, see *Sample Debugger Session* on page 31.

Flash Programming

In this chapter, we will learn how to write the Hello Flash application into the target flash memory in the context o µVision3 . The process of creation, setting compilation options, and building the application is described in detail in section "Hello FLASH" application for MCB2100.

The Selected flash programmer, Signum Systems JTAGjet Driver, must be properly configured before flash programming is performed for the first time. In the Options for Target dialog box, click the Settings button in the Utilities tab. The Flash Download Setup dialog box (Figure 20), appears prompting you to select the:

- Programming options (erase chip or sectors, write to flash, verify).
- Programming algorithm(s) to be used, depending on kind of the installed flash memory.
- RAM area used for programming algorithms.

This dialog box appears automatically if an attempt is made to program the Flash without configuring the programmer prior to that.

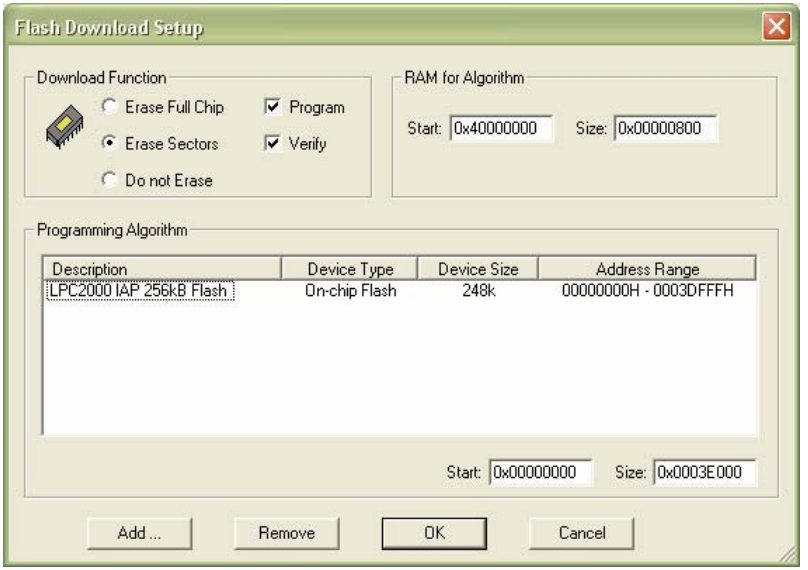


FIGURE 20 Configuring the flash programmer.

The Signum JTAGjet Driver flash programmer uses flash programming algorithms that come wit µVision3. The JTAGjet Driver programmer does not provide its own flash programming algorithms. Consequently, if µVision3 does not support programming of some specific types of flash memory, these flash memories cannot be programmed using JTAGjet either.

By default, Flash Download Setup tries to retrieve flash programming algorithm(s) associated with the selected processor's on-chip flash memory. The association is read fro µVision3's device database. Sometimes the association cannot be retrieved, in other cases our intention is to program the external (off-chip) flash memory located on the target board, rather than the internal, on-chip flash. In such cases, no flash programming algorithm is displayed initially in the dialog box or displayed algorithm is irrelevant. The user must select manually flash programming algorithms from those provided by µVision3.

In the Flash Download Setup dialog box, click the Add button to display a list of the available flash programming algorithms. The Add Flash Programming dialog box appears. Click Add (Figure 21).

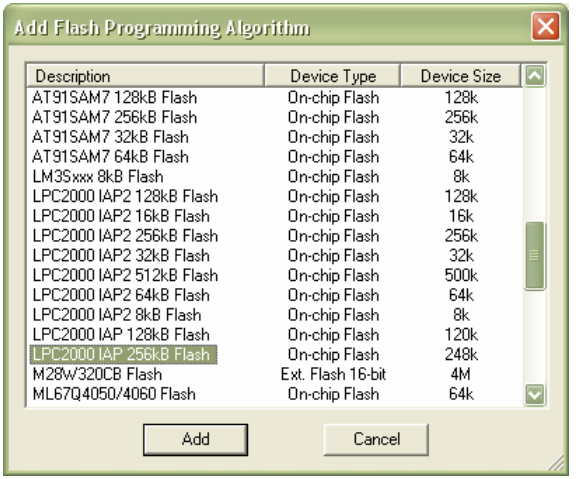


FIGURE 21 Manual selection of the µVision3 flash programming algorithm.

If more than one programming algorithm is to be used, repeat the selection process. The algorithms already selected can be removed, if necessary, using the Remove button of the Flash Download Setup dialog box.

By default, the area of RAM designated for flash programming algorithms is configured as the entire on-chip RAM memory. Modify the RAM starting address and the size as needed. Keep in mind that incorrect settings will prevent the flash programming algorithm from loaded successfully, causing flash programming to fail.

When the flash programmer has been configured, load the application code into the flash memory by selecting Download from the Flash menu (Figure 22).

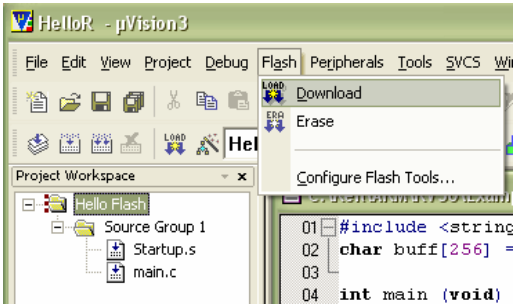


FIGURE 22 Programming flash memory from the µVision3 Flash menu.

The Flash menu contains these three menu items:

- **Download** — starts the flash programming operation with the current project target application.
- **Erase** — erases the entire flash memory.
- **Configure Flash Tools** — opens the „Options for Target" dialog box already opened in the Utilities tab for easy access to the flash programmer settings.

From the Flash menu, select Download. The JTAGjet emulator connects to the target and writes the application code to the LPC2129 on-chip flash memory. The messages displayed in the Build tab of the Output window confirm that the flash memory was erased and then programmed successfully with the HelloF.AXF application (Figure 23).

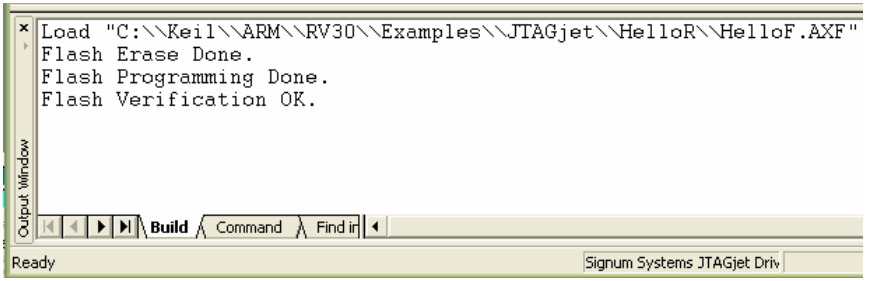


FIGURE 23 The Build tab of the Output window. Flash programming messages.

If the JTAGjet Driver emulation parameters were not configured prior to an attempt to connect to the target, the driver displays the „ARM Target Driver Setup" dialog box that prompts you to configure the driver (Figure 24). The

JTAGjet driver configuration process is described in section Configuring theJTAGjet ARM Driver on page 25.

Configuring the JTAGjet ARM Driver

Let us take a closer look at the process of configuring the JTAGjet ARM Driver using the ARM Target Driver Setup dialog box (FIGURE 24).

The ARM Target Driver Setup dialog box can be displayed by clicking Settings button located on Debug tab of Options for Target dialog box. Recall that it is also displayed automatically when the driver emulation parameters are not configured before the JTAGjet attempts to connect to a target board.

The driver emulation parameters, when set in the dialog box, are saved in two files located in the project folder: SigUV3Arm.cnf and SigUV3ArmJtagChain.cnf. The saved settings are used when µVision3 opens your project again.

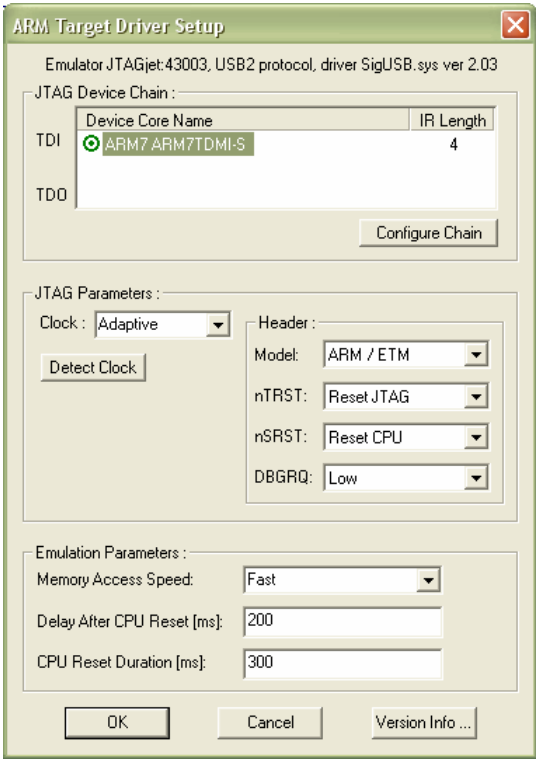


FIGURE 24 Setting the JTAG driver emulation parameters in the ARM Target Driver Setup dialog box.

The settings in the ARM Target Driver Setup dialog box are grouped into the following sections:

- **JTAG Device Chain** — Configuration of the JTAG device chain.
- **JTAG Parameters** — Selection of the JTAG clock frequency.
- **Header** — Selection of the JTAG connector type and behavior of the nTRST, nSRST and DBGRQ connector pins.
- **Emulation Parameters** — Other emulation parameters: the memory access speed, the delay after a CPU reset and the CPU reset duration.

The header selected in the JTAG Parameters Header section must match the JTAG connector used. The default is the 20 pin ARM connector available on

almost all target boards equipped with ARM microcontrollers. The default nTRST, nSRST and DBGRQ pin settings should not be modified without good reason.

The JTAG Parameters Clock section of the dialog box allows you to modify the JTAG clock (TCK) speed. The default is the frequency determined automatically while connecting to the target board. However, some ARM devices may need to have this changed to other values. Determine if adaptive clock should be used. It is recommended to manually test and select the JTAG clock frequency. To accomplish this, push the Detect Clock button and make the necessary changes in a new JTAG Configuration dialog box (Figure 25).

Configuring the JTAG Clock Frequency

The JTAG Configuration dialog box (Figure 25) allows you to manually test and select the JTAG clock frequency as well as the type of the JTAG header.

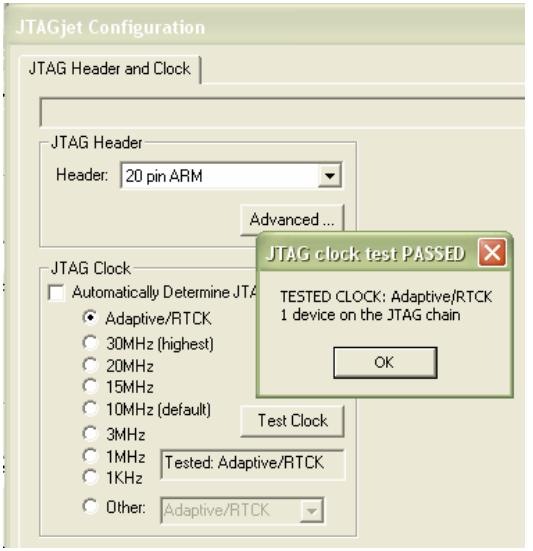


FIGURE 25 Testing and selecting the JTAG clock manually.

The JTAG clock (TCK) frequency displayed initially is the frequency determined automatically by the driver when connecting to the target board. Some ARM

devices, however, may require other settings. Determine if adaptive clock should be used. The Adaptive/RTCK option is reserved for those ARM devices that synchronize the JTAG clock with the CPU clock and return the RTCK clock signal back to the emulator. This is true for any ARM core with the -S suffix in its name. The Adaptive/RTCK option makes the JTAG channel functional across a broad range of the CPU speeds and controllable by the running application.

It is recommended to test the selected JTAG clock speed setting by clicking the Test Clock button. The program will communicate with the target board at the selected clock frequency and report errors, if any. Passing this test does not guarantee that the entire board will work at the given speed, since some memories may not be fast enough above the 10 MHz setting. However, if this clock test fails, than it is certain that the JTAGjet emulator will not able to neither communicate with the target and neither debugging nor flash programming session can be performed.

The Advanced button allows you to customize the pins on the JTAG cable. Extreme caution is advised when attempting to modify these settings. See Appendix for details.

Configuring the JTAG Chain

The driver always attempts to recognize the configuration of the JTAG device chain automatically when connecting to a target board. A list of the detected devices on the chain is displayed in the ARM Target Driver Setup dialog box starting from TDI at the top of the list to TDO at its bottom. If multiple devices are present, the user must select manually the ARM processor to be emulated. If only one device is present on the JTAG chain, then the selection is automatic. The selected device should be an ARM processor, or the driver will refuse to connect to it.

If automatic detection of the JTAG chain fails, or the devices are not recognized correctly, the JTAG chain can be configured manually. Click the Configure Chain button to open a new JTAG Configuration dialog box. The JTAG Header and Clock page allows you to select the JTAG clock frequency and the JTAG header. The JTAG Devices tab displays the detected devices on JTAG chain graphically (Figure 26).

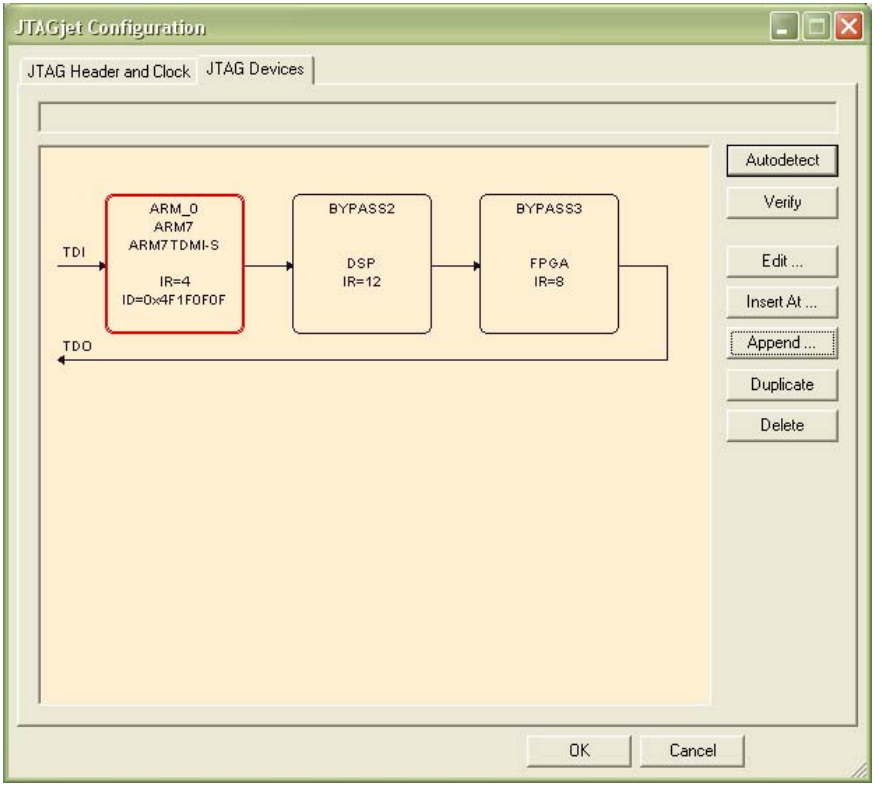


FIGURE 26 JTAGjet Configuration dialog box allows you to detect and configure the JTAG chain.

The JTAG devices on the chain are either detected automatically or imported from the last settings saved by the user. Press the Verify button to make sure that the scan chain is correctly defined and works with the target currently connected to the JTAGjet.

Use other available buttons to configure the JTAG chain properly or eliminate any settings errors detected with the Verify button. These buttons are:

- **Autodetect** —detects devices on JTAG chain automatically.
- **Edit** — allows you to modify the description of the selected device (in a separate dialog box). To select the device, click its symbol. The selected device is highlighted in red.

- **Insert At** — inserts new device in front of the selected device. The device parameters are set in a separate dialog box.
- **Append** — appends a new device at the end of the device chain (close to TDO). The device parameters are set in a separate dialog box.
- **Duplicate** — creates one more devices identical to the selected one.
- **Delete** — removes the selected device from the JTAG chain.

Special Settings

The ARM Target Driver Setup dialog box (Figure 24) allows you to configure the non-JTAG aspects of the ARM Target Driver, such as the memory access speed or the CPU Reset options. For now, accept the default settings by pressing the OK button.

At the time of this writing, the two other settings can be changed by modifying the SigUV3Arm.cnf configuration file stored in the project folder. These settings determine:

- if the CPU reset should be issued when the μVision3 debugger connects to the target. This setting is stored in the variable ResetCPUonStartup. The default is YES.
- if the values of the PC and CPSR ARM registers should be set to their default reset values after a CPU reset. This setting is stored in the variable SetPCandCPSRonCPUReset. The default is YES.

These settings are found in the [EmulationParameters] section of the SigUV3Arm.cnf configuration file. See the example below:

```
…
[Emulation Parameters]
BoardCfg=SigUV3ArmJtagChain.cfg
BoardDID=DEV0
BypassTest=no
JtagSpeed=0
JtagHeader=ARM,trst:c,dbgrq:0,reset:r
JtagInitDelay=200,r:300
MemFast=rw1
ResetCPUonStartup=yes
```

```
SetPCandCPSRonCPUReset=yes
…
```

When modifying the values of these parameters, keep the other settings in the file unchanged.

Sample Debugger Session

In this section we will go through two sample debugger sessions: one with the Hello RAM target application and the other with the Hello Flash target application. These applications are part of the HelloR project. See JTAGjet Debug Projects on page 11.

Debugging the Hello RAM Application

Recall that the Hello RAM application is loaded into the LPC2129 RAM at address 0x40000000. The program clears the global buffer buff[ ], then copies the string "Hello World" to the buffer, after which it enters an infinite loop that increments the local variable „i". In this section, we will learn how to run a µVision3 debugging session that takes us through the individual stages of that process.

1. Start a new debug session by clicking the Start/Stop Debug Session µVision3 toolbar button (Figure 27).

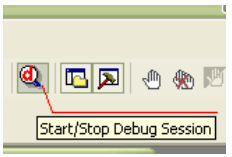


FIGURE 27 Starting a new µVision3 debug session.

µVision3 calls the JTAGjet ARM Target driver, which connects to the target board, loads HelloR.AXF into RAM at address 0x40000000, sets the PC to to this address, and executes the application from this startup point at the main( ) function.

2. From the View menu, open the Disassembly window and position the source code window with the main.c file in it next to the Disassembly window. Both windows show that code was executed until main( ) at address 0x400001AC (Figure 28).

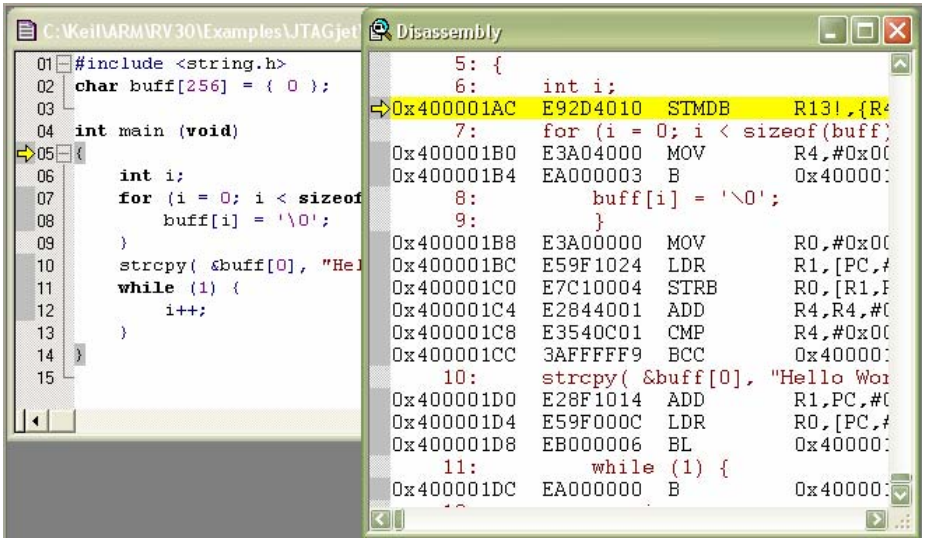


FIGURE 28 The Hello RAM application executed until the main( ) function.

3. In the main.c window, set a breakpoint on line 10 of the main.c module by positioning mouse cursor in the left margin of the window where mouse cursor changes its shape from vertical bar to cursor and double-click next to line 10. A red rectangle appears, confirming that the breakpoint was set (Figure 29).

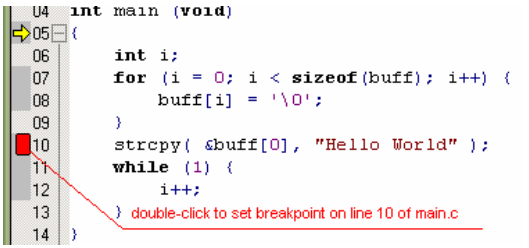


FIGURE 29 The µVision3 source window. Setting a breakpoint on line 10 of the main.c module.

4. Execute the application until the breakpoint by clicking the Run toolbar button. This fills the entire buff[ ] variable with zeros. A yellow arrow in Disassembly window shows the current PC position (line 10 in main.c and address 0x400001D0). The current values of the ARM CPU registers are also shown in the Registers tab of the Project Workspace window.

5. From the View menu, open the Memory window. The window appears at bottom of the main IDE window. Set the address of the Memory window to buff (global variable). This will position the window to show the memory area occupied by the buff[ ] variable. The buffer should be filled with zeros (Figure 31). The same variable can be also inspected in a Watch window.

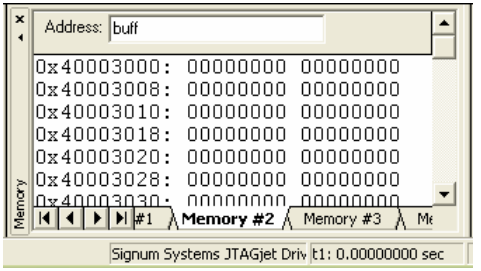


FIGURE 30 The memory window displaying the buff[ ] variable at address 0x40003000 as filled with null characters.

6. Set a second breakpoint on line 11 in the main.c module. Then run the application code once again. This time the code copying the string "Hello World" to the variable buff[ ] is executed. The application stops at line 11 in main.c (address 0x400001DC in Disassembly window). See Figure 32 below.

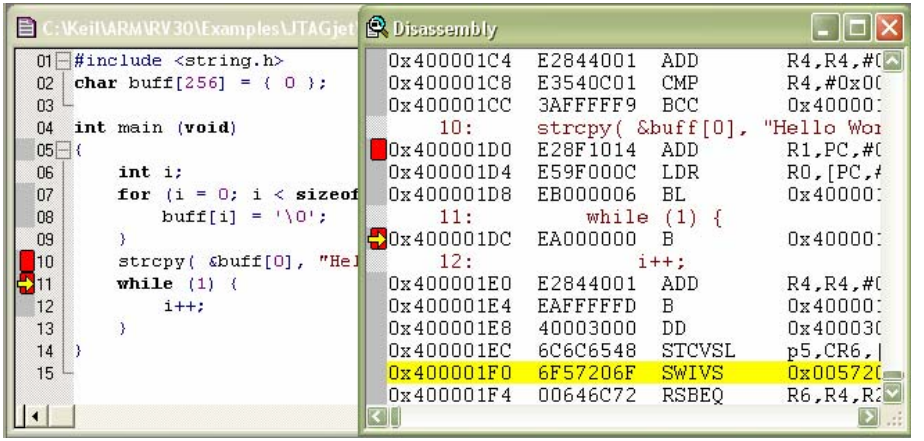


FIGURE 31 The application was executed until the breakpoint set at line 11 of the main.c module.

7. Verify that the contents of the buff[ ] variable in the Memory window has changed. Right-click in the window and select ASCII from a popup menu. The Memory window, now in the ASCII mode, shows that the string "Hello World" has been copied to the variable (Figure 32).

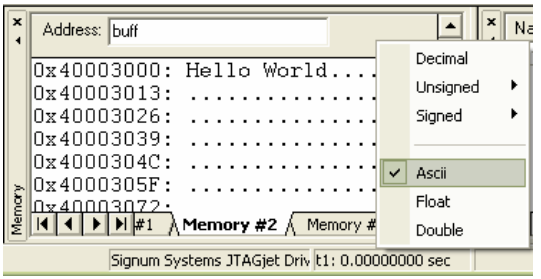


FIGURE 32 The buff[ ] variable, displayed in ASCII format, contains the "Hello World" string.

8. Finish the debug session by clicking the Start/Stop Debug Session µVision3 toolbar button again. µVision3 closes the Disassembly and Memory windows and returns to the project editing mode.

Debugging the Hello Flash Application

The Hello Flash application loads into the LPC2129 Flash at address 0x0. It performs the same actions as Hello RAM, but the code and data are located at different addresses. Namely, the code is placed at 0x0 and the data at 0x40000000. Since the code is loaded into the flash memory, only one hardware breakpoint can be set at a time, which steps from the limitations of the ARM architecture. As a rule, software breakpoints are not allowed in flash memory.

1. In the Select Target combo box on µVision3 toolbar Select Hello Flash as your current project target. Load the application into the flash memory by selecting Download from the Flash menu. The JTAGjet emulator connects to the target and writes the application code to the LPC2129 on-chip flash memory. Messages displayed in the Build tab of the Output window confirm that flash memory was erased and then programmed with HelloF.AXF application (Figure 23).
2. Start a debug session by clicking the Start/Stop Debug Session µVision3 toolbar button. µVision3 calls the JTAGjet ARM Target driver, which connects to the target board and then executes the application from the startup point at 0x0 till the main( ) function.
3. From the View menu, open the Disassembly window and position the source window with the file main.c in it beside the Disassembly window. Both windows show that code was executed until the function main( ) at address 0x000001AC (Figure 34).

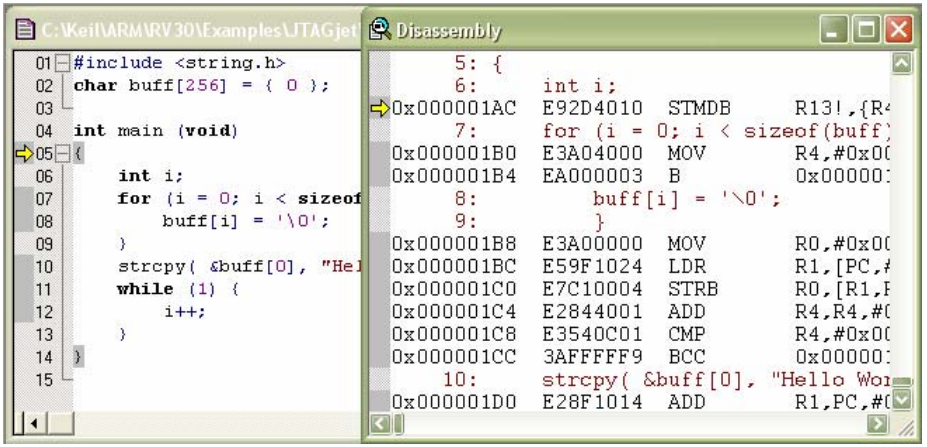


FIGURE 33 The Hello Flash application executed until C main( ) function.

4. Set the breakpoint on line 10 of main.c module by double-clicking in the left margin of the window next to line 10.

5. Execute the application until the breakpoint by clicking the Run toolbar button. This should fill the entire buff[ ] variable with zeros. The yellow arrow showing the current PC position now points at line 10 of the main.c module and address 0x000001D0 in the Disassembly window. The current values of ARM CPU registers are also shown in the Registers tab of the Project Workspace window.

6. From the View menu, open the Memory window. The window appears at the bottom of the main IDE window. Set the address of the Memory window to „buff" (global variable). This will position the window to show the memory area occupied by the buff[ ] variable, i.e., at address 0x40000000. The buffer should be filled with zeros (Figure 35). The same variable can also be inspected in the Watch window.

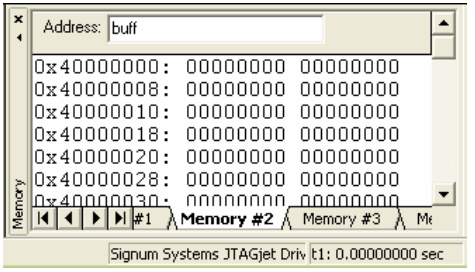


FIGURE 34 The Memory window showing the buff[ ] buffer at address 0x40000000 filled with null characters.

7. Make main.c source window active and click the Step over toolbar button to step over the current C statement. This executes the code on line 10, which copies the string "Hello World" to the variable buff[ ], after which it stops at line 11 (address 0x000001DC in the Disassembly window).

8. Verify the contents of buff[ ] in the Memory window. In the Memory window, right-click and in the popup menu, select ASCII. The Memory window shows, in the ASCII format, that string "Hello World" was copied into the variable (Figure 54).

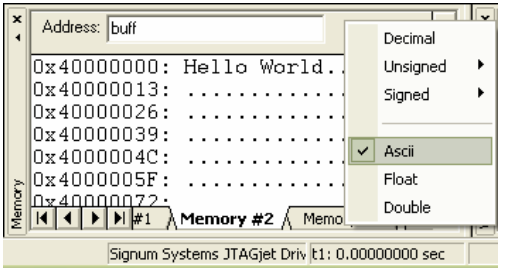


FIGURE 35 The buffer buff[ ] displayed in ASCII format with the string "Hello World" in it.

9. Finish the debug session by clicking once again the Start/Stop Debug Session toolbar button. µVision3 closes both the Disassembly and Memory windows and returns to the project-editing mode.

Working with JTAGjet-Trace

This chapter describes how to use JTAGjet-Trace with the Keil MCB2100 board. The board contains Philips LPC2129 ARM device which is based on the ARM7TDMI-S core.

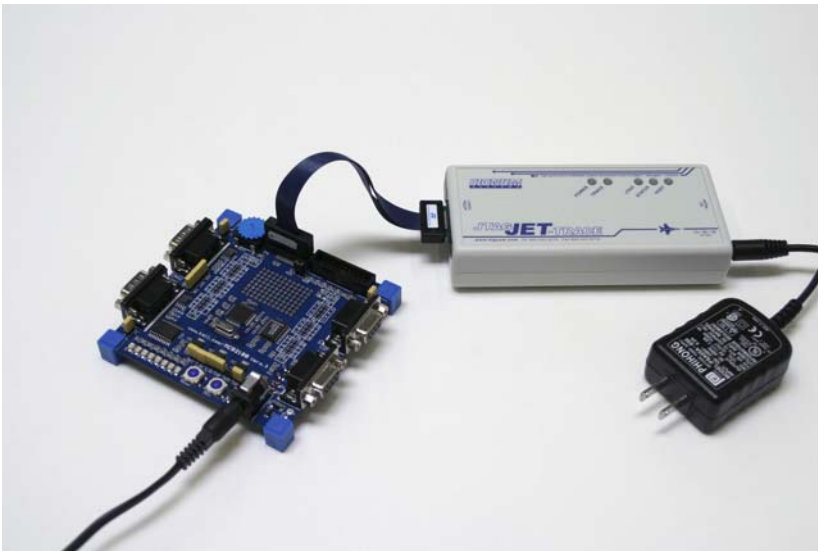


FIGURE 36 JTAGjet-Trace connected to the MCB2100 board.

Usually, these boards do not have the ETM connector installed, so you must do it yourself. The ETM connector is made by **Tyco** and the correct part number for a standard PCB (0.062 thickness) is: **767054-1**. These connectors are available form Signum Systems and from any Tyco distributor.

Caution

- Keil MCB2100 Rev 2.0 (and possibly earlier versions) board has a wrong pinout of ETM connector, making it impossible to use the ETM trace module with this board. DO NOT PLUG JTAGjet-Trace to this ETM connector. This problem has been fixed with Rev 3.0 of the board.
- Keil MCB2100 board shares the ETM pins with LEDs. If your application is blinking LEDs, these LEDs will not blink if ETM port is enabled.

JTAGjet-ETM Hardware Setup

It is assumed that the JTAGjet-Trace SOFTWARE and USB drivers were already installed. If not, install them before connecting to the target board.

1. Make sure you have the MCB2100 board revision 3.0 or later with ETM (P6) connector installed.
2. Verify that the following jumpers are installed:

   JP9 = ON (enables the JTAG port on Reset – factory default)
   JP8 = ON (enables the ETM port on Reset – not installed by factory)
3. Make sure target board is not powered. Connect the JTAGjet-Trace to the ETM connector on the target board using the blue ETM cable.
4. Connect power-supply to the JTAGjet-Trace. Power LED should come ON at this time.
5. Connect the USB cable to JTAGjet-Trace. The PC should automatically recognize the JTAGjet on the USB port.
6. Always power-up the target board last. The JTAG LED on JTAGjet-Trace should turn ON at this time, which indicates that target power has been detected and the H/W is ready to be used.

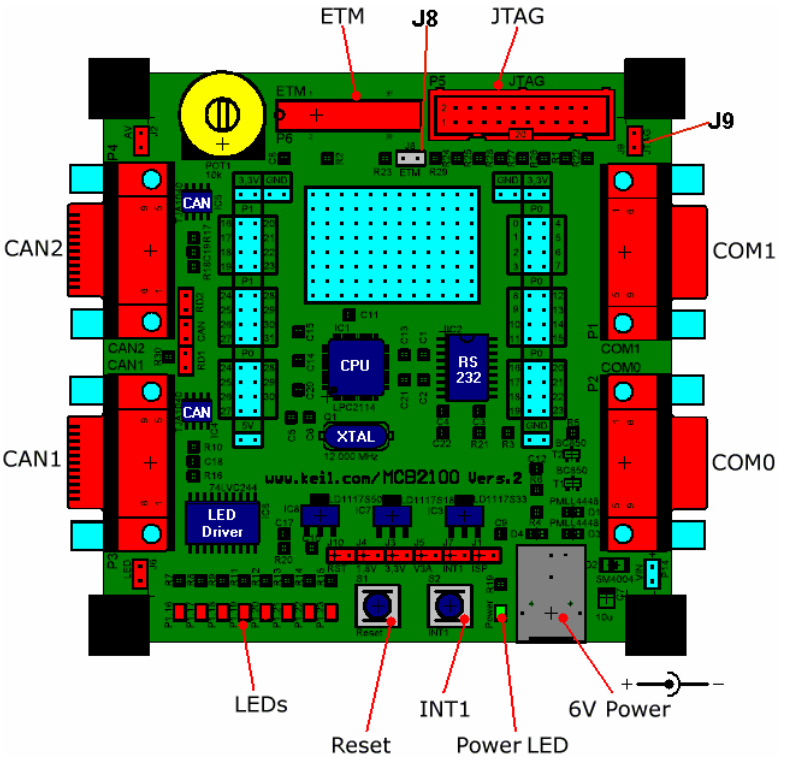


FIGURE 37

Note: It is strongly recommended to use USB 2.0 port. Complete trace buffer may consist of up to 18 MBytes of trace data and high-speed transfer is essential to avoid delays when reading the trace. USB 1.1 port will work 40 times slower then the USB 2.0.

µVision3 Debugger Setup

1. Execute the µVision3 debugger from the Windows Start menu. Open the Project - Open Project menu and browse to the

   `Keil\Arm\RV30\Boards\Keil\MCB2100`

folder. Select the Blinky.Uv2 project and click Open. In the Project menu, select Options for Target "MCB2100 RAM." Click on the Debug tab and select Signum Systems JTAGjet from the drop down menu.

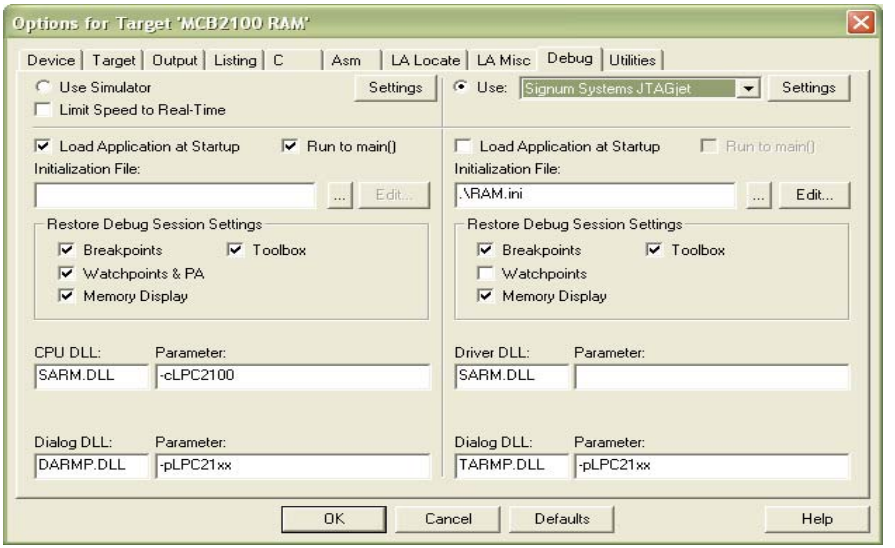


FIGURE 38 Selecting JTAGjet in the Options for Target dialog box.

Click OK

2. For this tutorial we will need the processor to stop before it gets to main(). In order to do this you need to click the Edit button in the Debug tab. When the text editor opens the RAM.ini file, comment out line:23 "g, main."

FIGURE 39 Editing the RAM.ini file

3. Most boards come with a 12MHz crystal. However, if your board has a 14.7 MHz crystal installed on it, in order for trace to work properly, you will need to modify the Startup.s file. Open the Startup.s file from the Project Workspace window, and click the Configuration Wizard tab in the bottom part of the window. Change the value of the PLL Multiplier Selection to 4 to accommodate the higher crystal frequency.

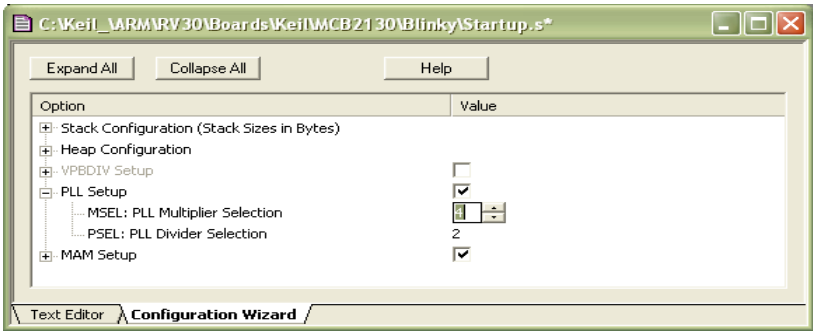


FIGURE 40 Modifying the PLL Multiplier.

Save the file.

4. In order to compile the project from the **Project menu** select **Rebuild all target files.**

5. Next, to start the debugger, select **Start/Stop Debug Session** from the Debug menu. The program should load into RAM, and the Debugger should open.

6. To open the trace window you need to select **Trace Window** from the Debug menu. When the Trace window is first opened the trace will be disabled. To enable trace, you need to click the **Control** button in the upper left hand corner of the Trace window. Next, in the **ETM Control** window you need to click on the **Enable ETM** check box.

Note This procedure needs to be repeated each time you start the uVision. Since the ETM pins are multiplexed with alternate CPU pin functions, leaving ETM port enabled on start would prevent the alternate functions from working. So it is up to the user to make a conscious action to enable the ETM trace whenever it is needed.

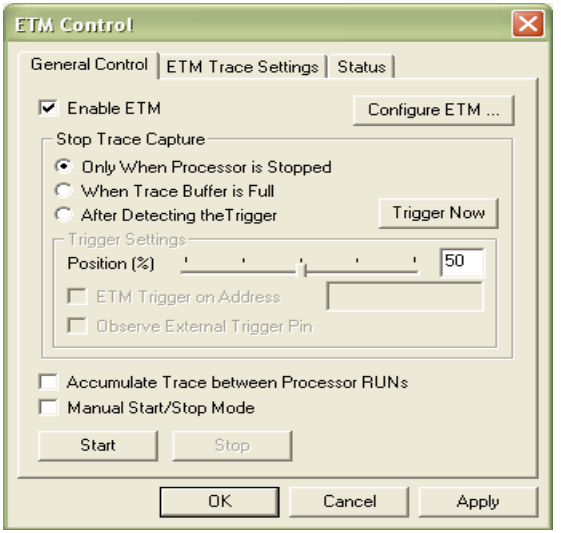


FIGURE 41 ETM Control, Enabling ETM

7. The **ETM Configuration** window will pop up. In this window you need to select the correct Board File, the **Keil MCB2100 LPC2xxx ETM Initialization** in this case.

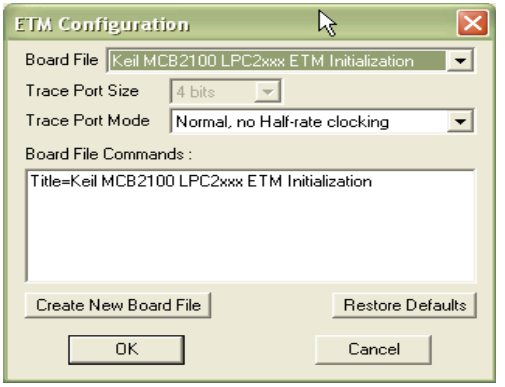


FIGURE 42 ETM Configuration

8. Click **OK** to close the ETM Configuration window. Selected settings are saved in the project configuration file and will be restored automatically next

time when the project is opened, so the step of configuring the ETM will be skipped when trace is enabled next time.

Working with ETM

1. Select the **Status** tab in the **ETM Control** window to display information related to the ETM resources embedded into the target ARM device. Please read the contents of the information dialog box and note the description of the ARM device's ETM resources (Figure 43). These resources can vary substantially from one ARM device family to another. Oftentimes, locating such information in the device's data sheet is cumbersome.

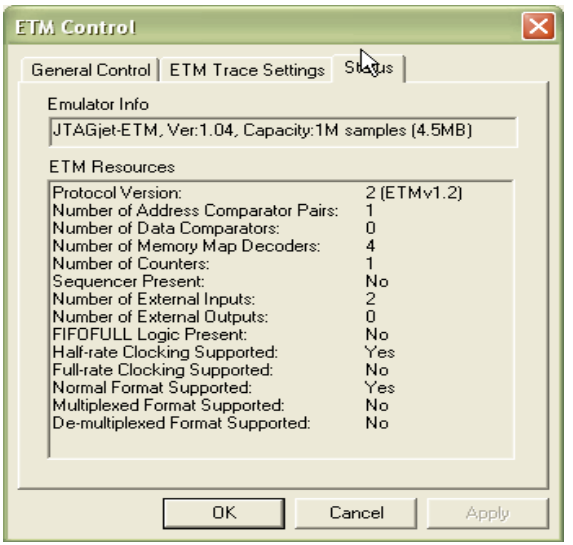


FIGURE 43 ETM Status tab showing the LPC2129 ETM resources

2. Select the **ETM Trace Settings** tab to make sure that the **Capture Complete PC Trace** is enabled and that the Data Tracing check boxes are cleared (Figure 44). Click **OK** to complete the trace configuration setup.

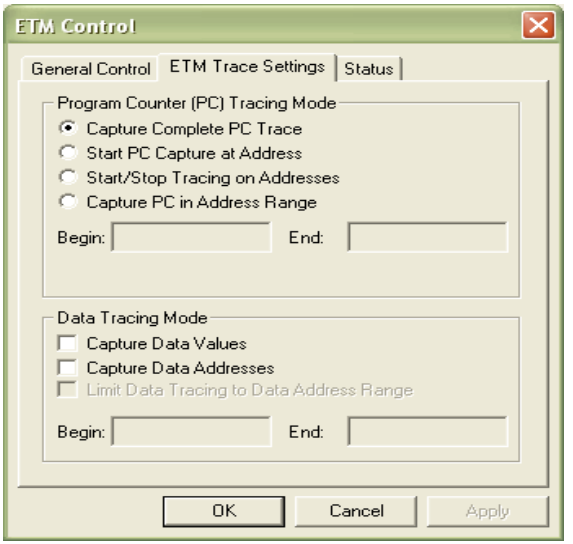


FIGURE 44 ETM Trace Settings tab.

Once the ETM is enabled, the Trace window will be cleared. The bottom status line shows now the current status of the trace; the Trace Clock frequency is approximately 60.00 MHz (Figure 45). The displayed frequency should be the actual CPU clock frequency.

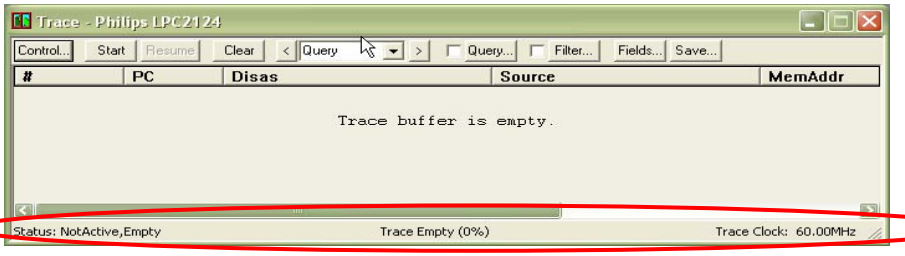


FIGURE 45 Trace Display Status bar showing activity, % usage and Trace Clock frequency

3. If the Trace Clock frequency in the Status bar displays 0, there is a problem with the target board, or the ETM connector is not plugged in all the way, or the jumpers are not installed properly. Please make sure the displayed frequency is correct for the application and **not more than 60 MHz** (silicon limitation) before continuing with the tutorial.

4. To capture the execution of the startup code only, open the **Disassembly** window, scroll down to address 0x40000314 `BEQ main(0x40000334)` and insert a breakpoint by double clicking on the leftmost column. A red square will appear, signifying the placement of a breakpoint. Now make sure that the Trace Window is open and click the **RUN** button on the tool bar to start program execution. When the breakpoint stops execution, the trace will display the end of the trace buffer, showing the last several executed instructions.

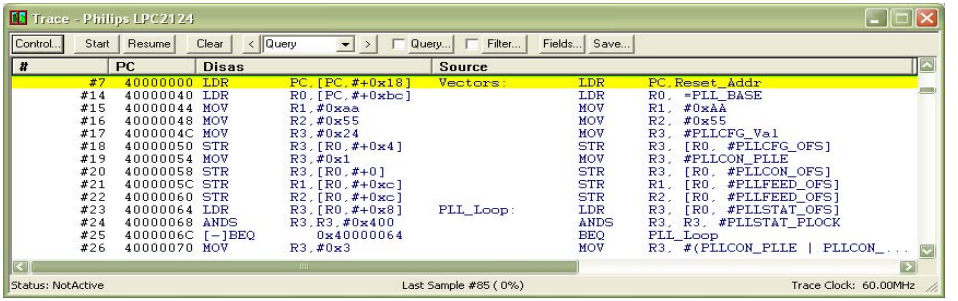


FIGURE 46 Trace Window showing the end of startup code written in assembly language

5. To fill the entire trace buffer with trace history, click the trace **Control** button and make sure the **Stop Trace Capture When Trace Buffer is Full** option is selected. Continue running the program by selecting **RUN** again. The trace buffer will accumulate a large amount of samples before it gets full. At that point, the Trace window will stop acquiring more data and begin displaying the captured information. Press the red **HALT** button on the toolbar to stop the CPU. This will allow the debugger to update the open memory and register windows.

6. **Synchronization with the Disassembly window** To see the trace buffer in the context of the loaded code, select a sample using the mouse or the arrow keys in the Trace window. The current trace sample will be highlighted in yellow. At the same time, in the Disassembly window a yellow right arrow will mark the code instruction that generated the highlighted trace sample. This synchronization between the Trace and Disassembly windows is referred to as CodeFocus.

7. **Trace Filtering** The ability to hide unwanted information from the trace buffer to get a clearer picture cannot be overestimated. Trace filtering is

easily accomplished by double-clicking in any column in the Trace window with the exception of the Time Stamp column. For example, to see only trace frames that contain C source lines, double-click inside any Source line column. In the menu that appears, select "**Filter non-empty Source**"

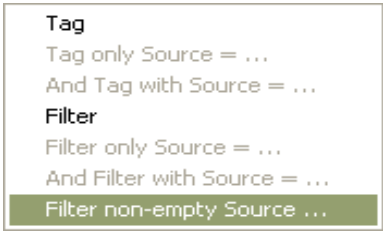


FIGURE 47 Filtering lines with non-empty source lines

The Trace window will filter out all frames without C source associated with them. With only lines of C code displayed, program execution will be easier to analyze.

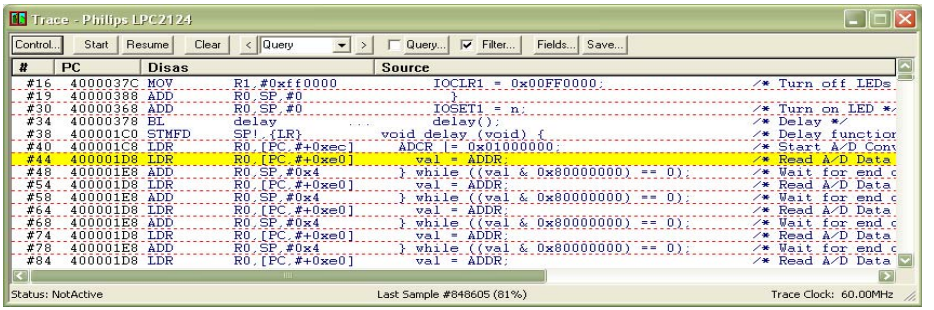


FIGURE 48 Filtering to show only the rows with C code

8. **Using Query** To return to the unfiltered view, uncheck the Filter check box on the toolbar. Click on the Query drop down list and change the query to look for the next **Function**. Keep pressing the forward [**>**] button until **int putchar(int ch) {** is found. Double click on the **PC** column of that trace line and from the menu select "**Filter only PC = 0x40000414**". This will change the filter to show only calls to **putchar**. This will allow us to measure the frequency of the calls and determine if the interval between calls is constant.

9. To find out how often the **putchar** function is called, scroll the trace window to the right to see the **TStamp** column and double-clicking on it until it shows the time in delta mode and cycles **[dt][cyc]**. At the end of the process, we find that the call takes place every 124160 CPU cycles. To convert this value to microseconds, click on the TStamp header and select the **usec** from the list.

   Now double click anywhere in the time stamp column to change the display to [abs][us]. Scroll to the end of the buffer to see how much execution time is covered by the contents of the trace buffer. A 256K buffer, the smallest available, is capable of storing over 11 msec worth of a program running at 30 MHz. With the largest trace buffer of 4M, this capability is 16-fold larger.

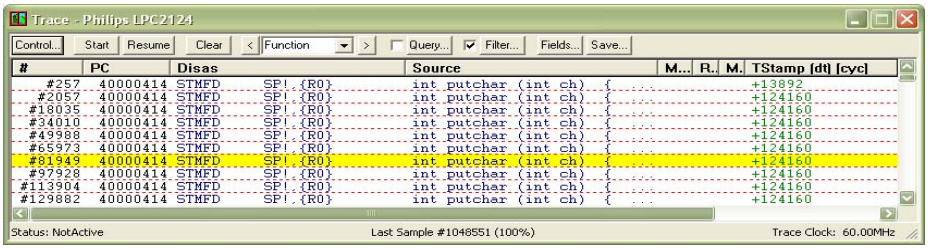


FIGURE 49 Filtered trace showing calls to only one function.

10. As you probably noticed, our experiment filled the trace buffer in an instant. This is because the buffer stored the samples indiscriminately without eliminating such uninteresting fragments of the application as loops waiting for ready bits or software delays. This can easily be amended by pre-filtering the data that comes into the trace buffer so that only the relevant information is retained. One way to do it is to set the trace to capture execution in PC address range that interests us.

    Please uncheck the **Filter** to see all samples, and then click on the **CLEAR** button to erase the trace buffer. Now open the Control dialog box and select the **ETM Trace Settings** tab.

    In the Settings tab, select the **Capture PC in Address Range** setting and enter the following addresses:

    Begin: 0x40000330 End: 0x400003ac

    These settings will capture execution trace of the main.c program loop only, and any function calls outside of main will not be recorded. This is a great

way to catch the profile of the application as you can see clearly the timing of the major system functions.

Click **OK** to save the settings. Restart the debugger by clicking **Start/Stop Debug Session** in the debug menu twice. Open the Trace window again, and make sure that Trace is enabled.

Click the **Run** button. Notice that the trace buffer fills much slower this time, as it needs to make selections in order to store only the lines in the main loop. When the trace fills up to about 2% stop the CPU by clicking the red **Halt** button and hit the Home button on your keyboard to see the beginning of the trace. The window should look similarly to that in Figure 50.

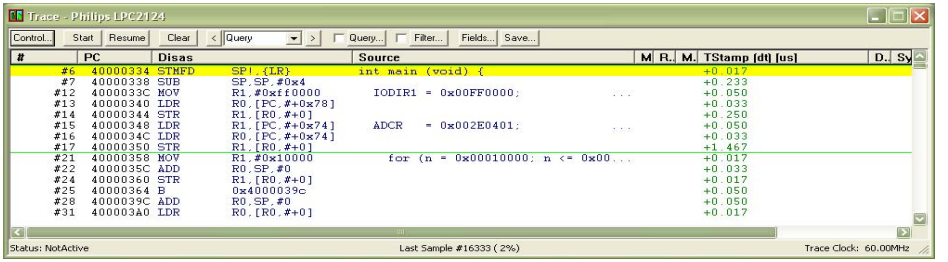


FIGURE 50 Filtering to show calls to one function only.

The green lines in the Trace window represent trace discontinuities. As expected, the trace started tracing from the first line of the main program (**main**) and stopped on 0x40000350, which is there where the first function (**init_serial()**) was called. We find that this function stopped the trace for 1.467 µs and started to capture again when it returned to the main program to execute the **for** loop. The trace captured a few lines from main and stopped again when **delay()** was called, which took 34085.792 us.

11. To eliminate any function from the trace, we must use the **Start/Stop Tracing on Address** feature. Also, it is necessary to enter the function's exit address in the trace **Begin** field and the function's entry point in the trace **End** field. In this example we enter 0x40000480 in the **Begin** field and 0x40000414 in **End** field. The address placed in the Begin field is the entry point of the putchar() function, and the address placed in the End filed is

the exit address of putchar(). Figure 51 shows an example with the putchar() function filtered out.

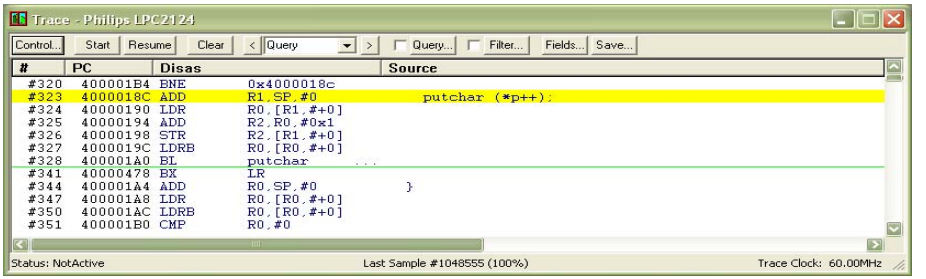


FIGURE 51 Trace with the putchar() function filtered out..

In the above figure you can see that where putchar() was called there is a green line signifying a trace discontinuity because the code inside of the function was filtered out.

Data Tracing

Tracing of code execution is an excellent method for detecting many programming errors. However, without an ability to visualize data transfers, it may be hard to see what the application is actually doing. In typical C code, most problematic places are prologue and epilogue of functions where several 32-bit CPU registers are transferred to or from the stack with the STMIA and LDMIA instructions.

The Philips LPC21xx data sheets do not mention the capability of data tracing over the ETM, which may suggest that it is not supported, most likely due to the lack of bandwidth on the 4-bit ETM.

Nevertheless, our experiments show that if data access is not very frequent, some data variables may be successfully traced even on a 4-bit ETM port. Frequent variable access will overflow the on-chip ETM FIFO, causing loss of PC information. Therefore tracing variables may require a certain amount of caution.

In the ETM Control dialog box, you can choose to trace the variable(s) address or data or both. Since each variable address is 32-bit long and most data contain 32-bit values, up to sixteen clock cycles may be used just to export each variable.

Thus, whenever possible, using byte variables will save a lot of bandwidth when tracing.

1. To trace what is read and written to the IO ports, click on the Control button, select the ETM Trace Settings tab and make sure the Capture Complete PC Range is selected (Figure 52).

   In the **Data Tracing** section right below, check all three boxes and enter the addresses **0xE0028000** and **0xE002801c** in the Begin and End fields, respectively.

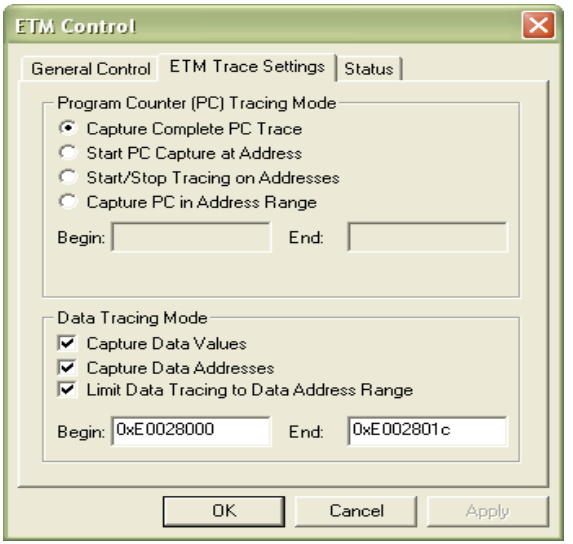


FIGURE 52 Adding tracing of variables to the ETM settings

2. Click **OK** to close the Control window. Restart the debugger by clicking **Start/Stop Debug Session** in the debug menu twice. Open the Trace window again, and make sure that Trace is enabled.

3. Press the **Run** button. The trace will fill instantly and display new data. To find the data variables in the trace, double click in the MemAddr or MemData fields and select Filter non-empty MemAddr. This will only display the trace frames with valid MemAddr fields. In our exercise, only the writes and reads to the IO ports and the associated data values are shown (Figure 53).

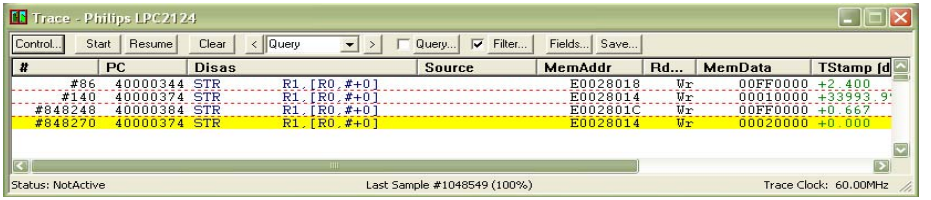


FIGURE 53 Filtered trace showing only read and write to the IO ports operations.

Using the Logic Analyzer Window

The µVision3 debugger allows monitoring program variables during program runtime using a configurable **Logic Analyzer** window. To open the window, select Logic Analyzer Window from the View menu or press the Logic Analyzer Window toolbar button.

The JTAGjet ARM Driver for µVision3 allows simultaneous monitoring in the Logic Analyzer window of up to two program variables. This is accomplished with the assistance of the ARM CPU hardware breakpoint/watchpoint registers. These can be used as ordinary hardware breakpoints[1] or as watchpoints to catch write operations at specific memory addresses. When the memory location whose address has been loaded into the Watchpoint register is written, the CPU stops. The value of the variable is then read and program execution resumes. As this method stops the CPU for a few milliseconds only, it can be used safely whenever the program does not depend heavily on external real-time events. The Logic Analyzer window's limitation on the number of the monitored variables stems from the fact that the ARM7 and ARM9 processors are equipped with only two hardware breakpoint/watchpoint registers.

To configure the Logic Analyzer window for monitoring variables, press the window Setup button and add the names of the variables in the **Setup Logic Analyzer** dialog box that appears (Figure 54).

[1] Hardware breakpoints can be set in Flash memory.

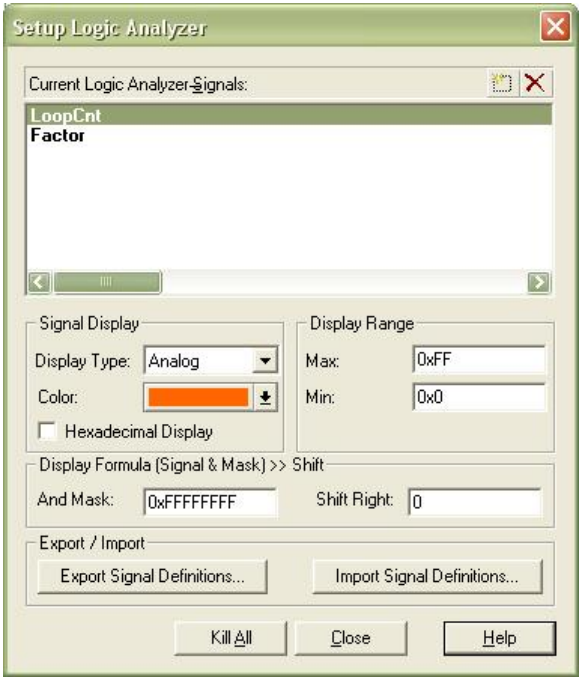


FIGURE 54 Adding variable names for monitoring in the Logic Analyzer window.

Complete the setup and close the dialog box.

Remove or disable all breakpoints currently set and run the program. The **"Logic Analyzer started …"** message is displayed in the µVision3 Output window. The Logic Analyzer window begins displaying the monitored variables as they are modified during program execution.

To stop variable monitoring, stop the CPU. The µVision3 Output window displays the **"Logic Analyzer stopped (nnn records sent)"** message. A typical graph plotted for two variables, `LoopCnt` and `Factor`, is shown in Figure 55.

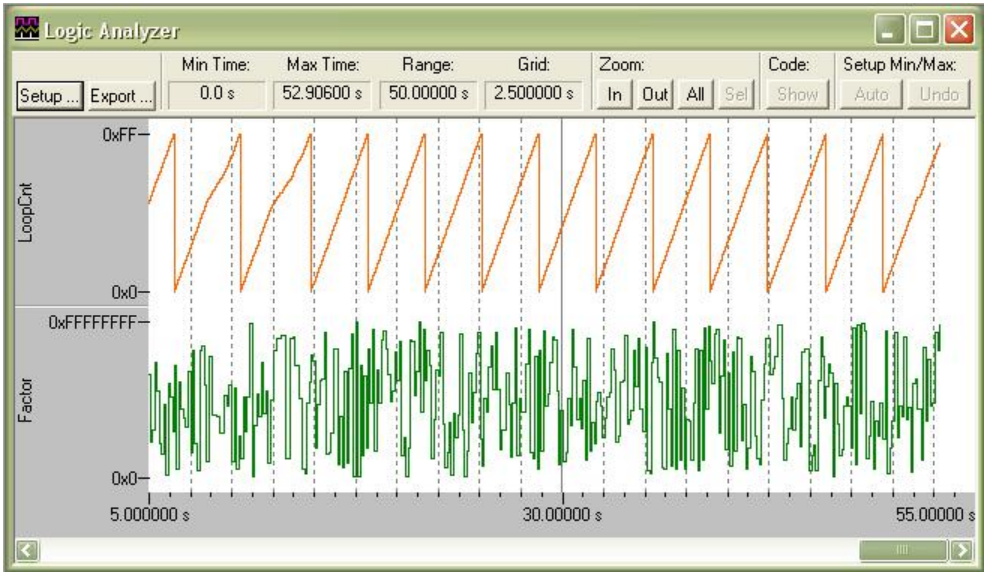


FIGURE 55 Example graph plotted in the Logic Analyzer window.

The Logic Analyzer window monitors variables only when your program runs freely, that is, when it is executed using the Run command.

Variables are not monitored when stepping through the program (actions Step Into, Step Over and Step Out) or when executing it with the active breakpoints. When the Logic Analyzer window is set up for monitoring variables and the CPU is run using the active breakpoints, the µVision3 Output window displays the message **"!!! Breakpoints are in use – Logic Analyzer not started. Disable breakpoints to use Logic Analyzer."**

Variables are also not monitored when the Logic Analyzer window is closed, even if the CPU runs freely and the window is setup for variable monitoring. This prevents stopping the CPU momentarily for reading the values of modified variables when the window is closed.

For more details, please refer to the description of the Logic Analyzer window in the µVision3 documentation.

Appendix

Advanced JTAG Configuration

As mentioned in section *Configuring theJTAGjet ARM Driver* on page 25, the ARM Target Driver Setup program allows you to customize the JTAG cable pin assignment and characteristics. To customize the pins further, click the Configure Chain button or the Detect Clock button and then select Advanced in resulting JTAG Configuration dialog box. This brings up the Advanced JTAG Configuration dialog box (Figure 54).

Extreme caution should be exercised when reconfiguring the JTAG cable pins. Make pinout modifications only if there are problems connecting to the target board, or if a specific target board requirement must be met.

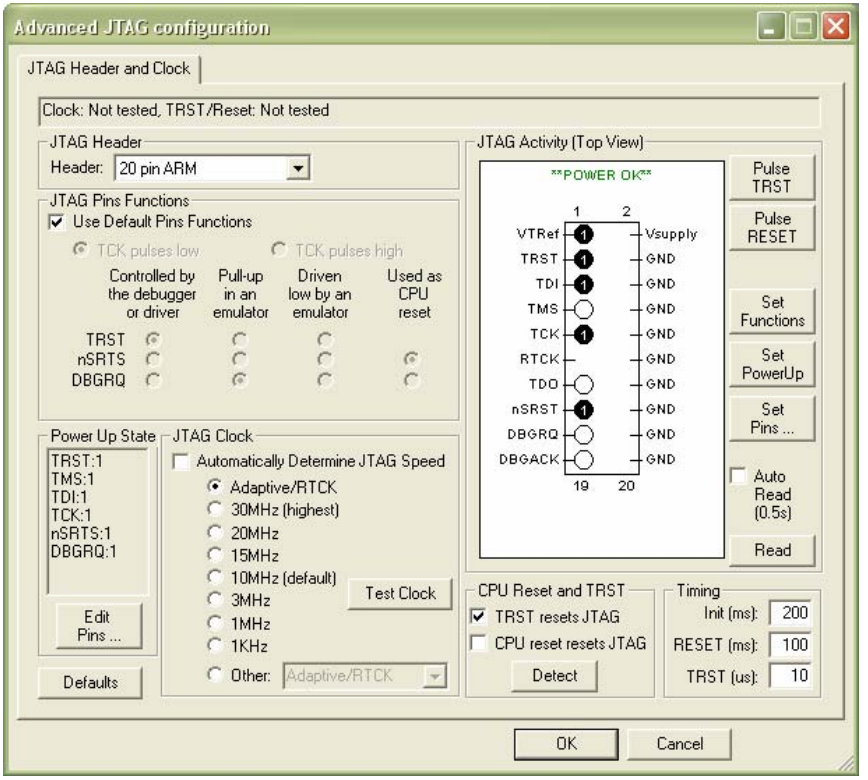


FIGURE 56 The advanced JTAG Configuration dialog box.

The Advanced part of the JTAG Header and Clock tab displays the status of the JTAG header along with controls determining how the JTAG pins are to be terminated or set when the target is powered-up. The dialog box also allows you to change any of the JTAG pins, which may be useful when troubleshooting JTAG chains on new boards.

Troubleshooting

1. Make sure that your JTAGjet emulator uses its latest hardware image. The Emulator Diagnostic Utility, EmuDiag, enables you to update the

JTAGjet firmware using the procedure described in the EmuDiag Help file. EmuDiag can be downloaded from www.signum.com/support.htm.

2. Use the EmuDiag.EXE program to test the JTAG connection.
3. Make sure that the JTAG clock is stable, and its speed does not exceed the recommended value. Recall that the JTAG clock should not be faster than 1/3 of the current CPU frequency. As some CPUs start up at a very low CPU clock speed, verify the lowest CPU clock rate and set the JTAG clock to 1/3 of that rate.
4. For ARM cores with the name suffix –S, select Adaptive/RTCK clock. Otherwise, the driver may not communicate reliably with the target board.
5. Use AutoDetect and Verify in the JTAGjet Configuration dialog box (Figure 26) to test and verify your scan chain.
6. Make sure that the CPU memory and PLL are correctly configured. Do that by verifying your application startup code and, if necessary, by preparing the correct board script file *.INI executed by µVision3 when connecting to the target.

❑