Getting Started with Freescale MQX™ RTOS and MDK-ARM Keil™ µVision4®

PRODUCT:	Freescale MQX™ RTOS
PRODUCT VERSION:	Freescale MQX 4.0.2
DESCRIPTION:	Using MDK-ARM Keil™ µVision4® Tools with Freescale MQX™ RTOS
RELEASE DATE:	August, 2013

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

Document Number: MQXGSKEIL
Rev. 02, 08/2013

Table of Contents

1 Read Me First

This document describes steps required to configure the MDK-ARM Keil™ µVision4® development tools and use it to build, run, and debug applications of the Freescale MQX RTOS operating system. See the *Getting Started with Freescale MQX™ RTOS* and other user documentation included within the latest Freescale MQX RTOS installation for more details not specifically related to ARM® development tools.

Get the latest Freescale MQX RTOS at freescale.com/mqx.

Additionally, you can find more information related to Freescale Kinetis platform support in µVision4 tools in the arm.com/files/pdf/Kinetis_LAB.pdf document.

2 Building the MQX Libraries

2.1 Compile-time Configuration

Major compile-time configuration options are centralized in a single user configuration file located in

```
<install_dir>/config/<board>/user_config.h
```

This user configuration file is included internally together with the private configuration files in MQX PSP and BSP.

To share configuration settings between different boards, the user_config.h file may include other header files with common settings. The header files may only be located in the same <board> directory or in the “common” directory:

```
<install_dir>/config/common
```

All MQX configuration files are also *indirectly* used by other core components like RTCS, MFS, Shell, etc. “Indirectly” means that the MQX PSP and BSP must be built first, which causes the configuration file to be copied into the output (lib) directory. The other components, then, include the configuration file from the `/lib` output directory.

Caution: Until the PSP or BSP libraries are rebuilt, configuration changes made in the `user_config.h` file are not used by any other MQX component. On the other hand, after the PSP and BSP libraries are re-compiled with a new configuration, it is important to recompile the other libraries so the compiled code is consistent with the configuration file. See the next section for more details.

2.1.1 Build Process

After any change to the compile-time user configuration file or MQX kernel source files, the MQX libraries need to be re-built. The build process is similar for all core components:

The output directory for any MQX library component is `<install_dir>/lib/<board>.<compiler>/<component>`

For example the MQX PSP and BSP libraries for the TWR-K60N512 board are copied into the `/lib/twrk60n512.uv4/psp` and `/lib/twrk60n512.uv4/bsp` directories after successful build process.

All public header files needed by an application to make use of the library are also copied from internal include folders to the same output directory.

During PSP or BSP build process, also the `user_config.h` file and other header files from the `config/<board>` and `config/common` directories are copied into the `lib/<board>.uv4` output directory.

Other components like RTCS, MFS, Shell or USB use the copied configuration files only.

Applications which make use of any MQX library do not need to make any reference to the internal source and include paths of the MQX components. Applications use solely the paths in the `/lib/<board>.<compiler>` as the search paths for header files or libraries.

To summarize the points above, follow the guidelines below when re-building the MQX libraries:

After any change to the `/config/common/user_config.h` file, all MQX libraries should be re-built.

The PSP and BSP libraries must be build first, before the MFS, RTCS, USB, Shell, and other libraries.

Important: No changes should be made to header files in the output build directory (`/lib`). The files get overwritten any time the libraries are built.

2.2 Build Configurations

Each µVision4 project in Freescale MQX RTOS contains multiple compiler/linker configurations (so called build „targets").

Two different types of build targets exist for different compiler optimization settings:

Debug – the compiler optimizations are turned off or set to low. The compiled code is easy to debug but may be less effective and much larger than the Release build. All output libraries have **_d** postfix in the file name (e.g. `rtcs_<board>_d.a`).

Release – the compiler optimizations are set to maximum. The compiled code is very hard to debug and should be used for final applications only. There is no postfix in the output file name (e.g. `rtcs_<board>.a`).

Build target name of any MQX application project makes a reference either to **Debug** or **Release** builds of the core libraries. On top of that, the target names also specify board memory configuration which gets built. For example:

Devices with internal Flash memory (e.g. TWR-K60N512):

Int. Flash Release – this target is suitable for final application deployment. When programmed to Flash, the application starts immediately after reset. Variables are allocated in internal SRAM memory.

Int. Flash Debug – same as above, only the Debug-compiled libraries are used. This target is suitable for debugging before deployment. On boards without external memory, this is the only target suitable for debugging larger applications.

Boards and devices with internal Flash memory and additional external RAM for data (TWR-K70F120M):

Int Flash <mem>Data Debug – The name of each target additionally defines a memory used as the default data storage. For example, the application built with target named "Int Flash DDRData Debug" will execute code out of internal Flash memory and will use the DDR memory for data storage.

Boards with external RAM memory:

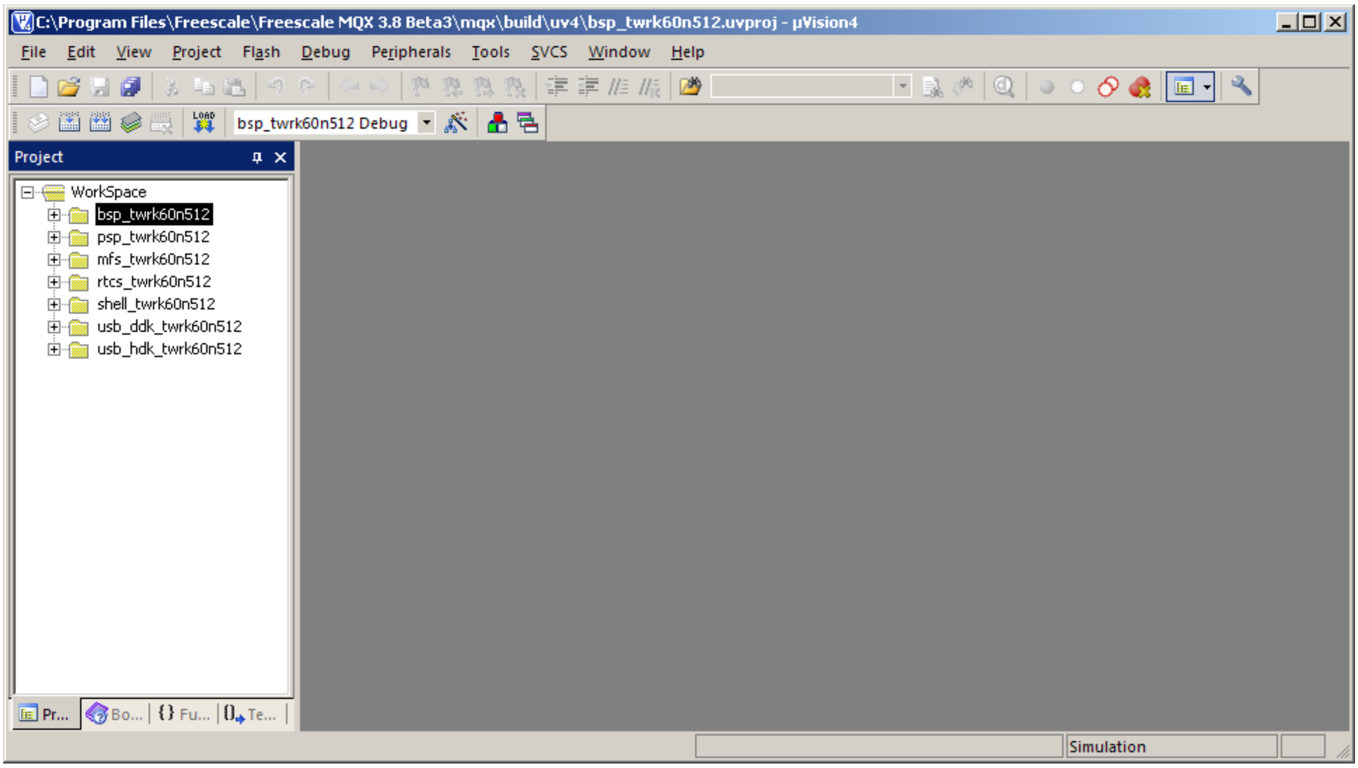
Ext. Ram Debug – solely for debugging purposes with code located in external RAM memory. Both code and variables are located in this external memory. Application executable is loaded to RAM automatically by the debugger.

See the BSP-specific information included in the latest MQX installation for a description of build targets specific to particular board.

2.3 Batch Build in µVision4 IDE

With µVision4, the MQX build process can be simplified by using Batch Build feature. For each supported board, there is a Multi-Project Workspace file which includes build projects for all related MQX libraries:

`<install_dir>/config/<board>/uv4/build_libs.uvmpw`



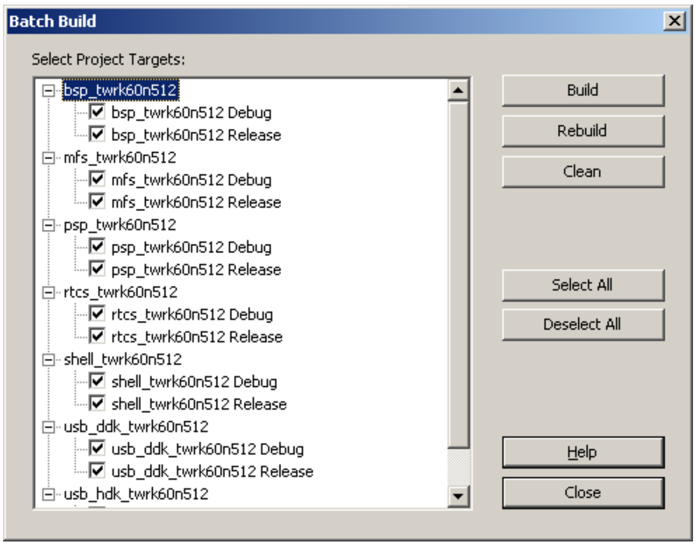
The Workspace file contains Batch Build configurations which can be used to build all MQX libraries at once.

Go to menu "Project / Batch Build..." in the µVision4 IDE.

Select libraries and targets to build. Note that the projects are sorted in alphabetical order but this does not affect the build order. The build order is set properly in the Multi-Project Workspace definition.

Select the libraries and targets you want to build in a batch. It is recommended to rebuild at least all Debug or all Release targets at once.

Press the "Rebuild" button to start the batch build process.


Batch Build
Select Project Targets:
bsp_twrk60n512
bsp_twrk60n512 Debug
bsp_twrk60n512 Release
mfs_twrk60n512
mfs_twrk60n512 Debug
mfs_twrk60n512 Release
psp_twrk60n512
psp_twrk60n512 Debug
psp_twrk60n512 Release
rtcs_twrk60n512
rtcs_twrk60n512 Debug
rtcs_twrk60n512 Release
shell_twrk60n512
shell_twrk60n512 Debug
shell_twrk60n512 Release
usb_ddk_twrk60n512
usb_ddk_twrk60n512 Debug
usb_ddk_twrk60n512 Release
usb_hdk_twrk60n512
Build
Rebuild
Clean
Select All
Deselect All
Help
Close

3 MQX Task Aware Debugging

MQX Task Aware Debugging plug-in (TAD) is an optional extension to a debugger tool which helps to visualize internal MQX data structures, task-specific information, I/O device drivers, and other MQX context data.

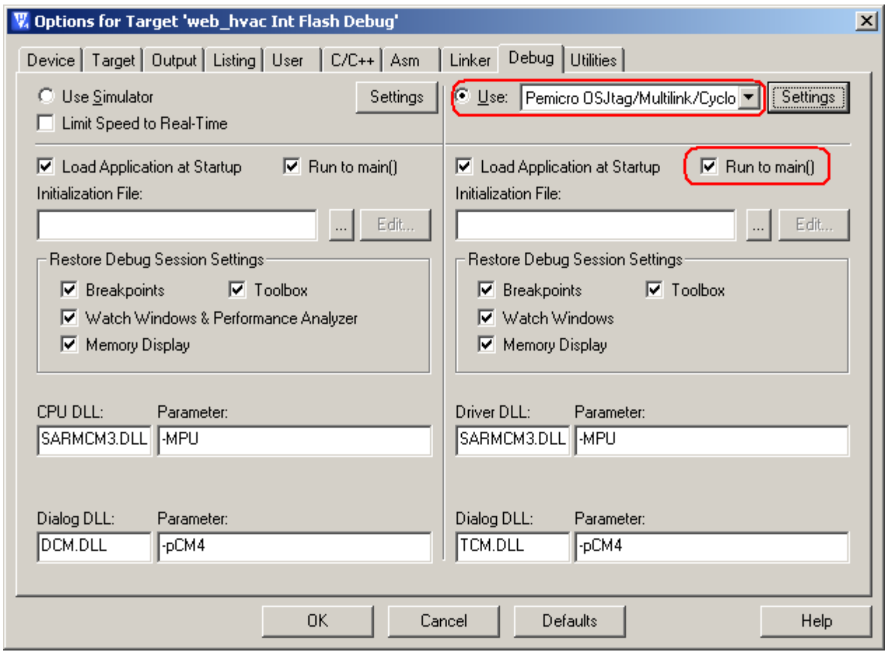
The MQX TAD plugin for µVision4 is called MQX-Viewer and is installed by an add-on installer included with the MQX installation.

3.1 Debugging MQX Applications in µVision4

Loading and debugging MQX applications is an easy task with ARM Keil µVision4 tool and it is not really different from debugging classic non-OS applications. Ensure that you select the correct debugger interface in the project options and correct processor configuration.

3.1.1 Using OSJTAG Debugger Connection

The MQX example projects are configured by default to use on-board OSJTAG debugger connection. You can double check the debugger connection settings in project options, the “Debug” tab:

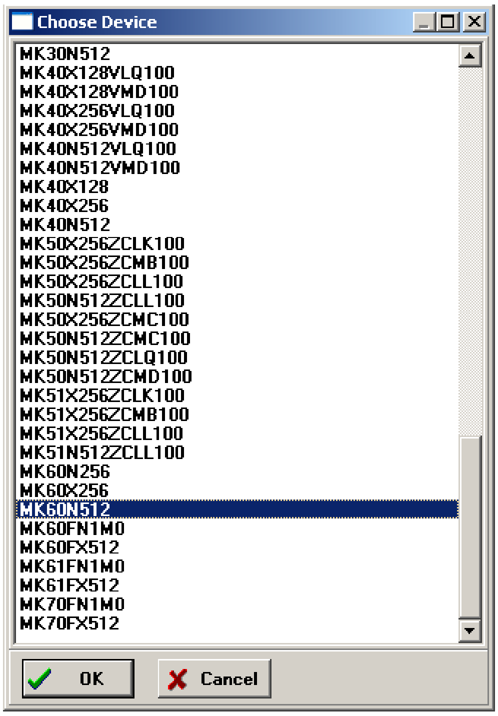


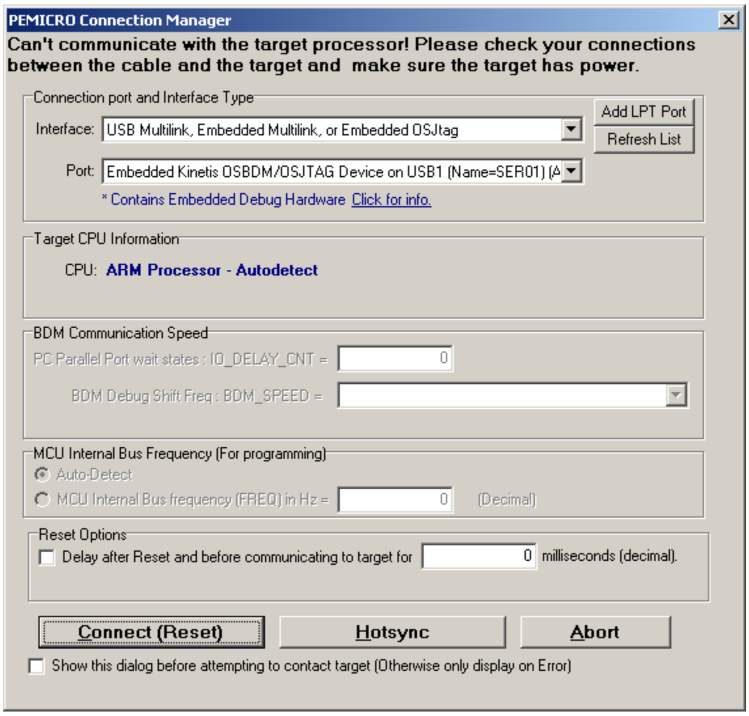
You may also want to select the “Run to main()” option which forces the debugger to execute the startup code until the main C entry point function is reached. When this option remains unchecked, the debugger stops execution at the first executed instruction at the reset vector.

In either case, be aware that, at such breakpoint, the MQX Operating System is not yet fully running so use of TAD plugin features, as described in subsequent sections, is limited.

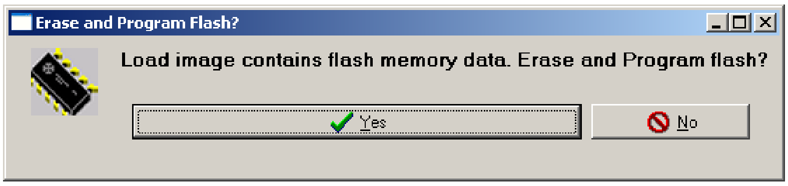
When a MQX application is compiled and linked with all MQX libraries, press the “Start/Stop Debug Session...”  button on the toolbar to initiate the debugger session.

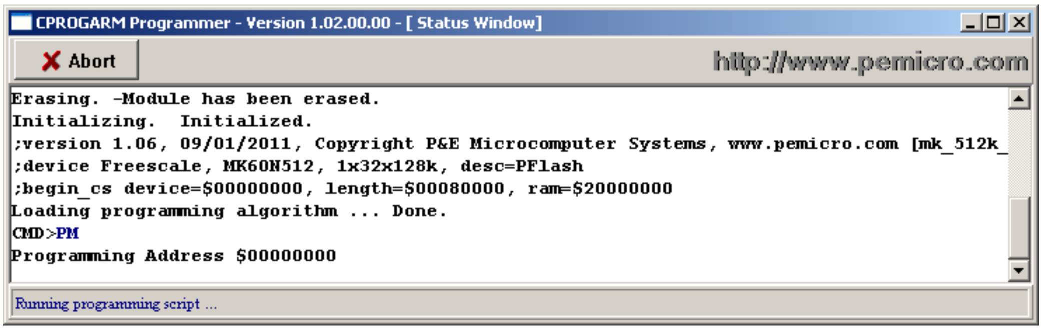
With the OSJTAG debugger connection, the first time you try to execute the application you will be prompted to select the target processor. Then, in the connection manager, press the Connect (Reset) button to establish connection between the PC Host and target board.



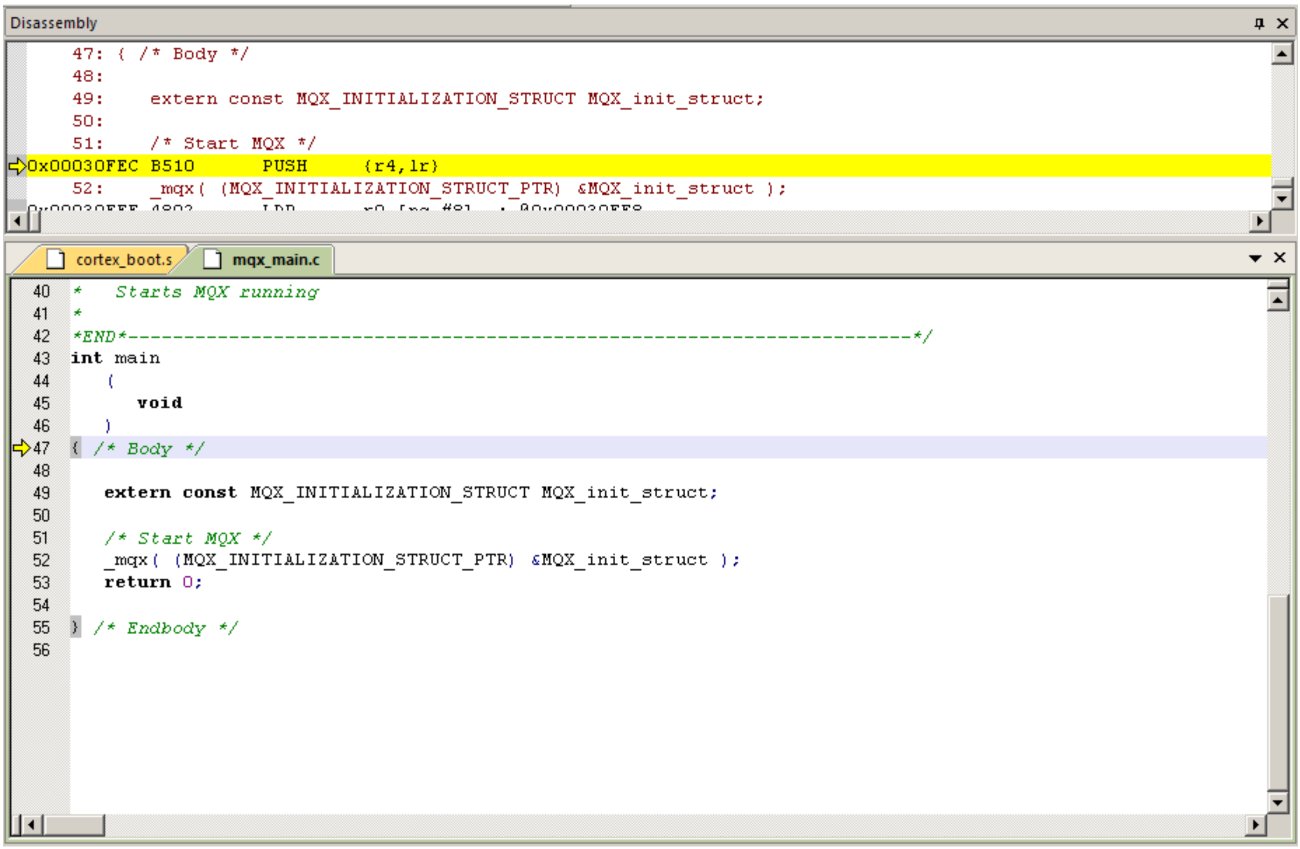


Use the “Yes” button to load application to target Flash memory and wait until the Flashing process finishes:





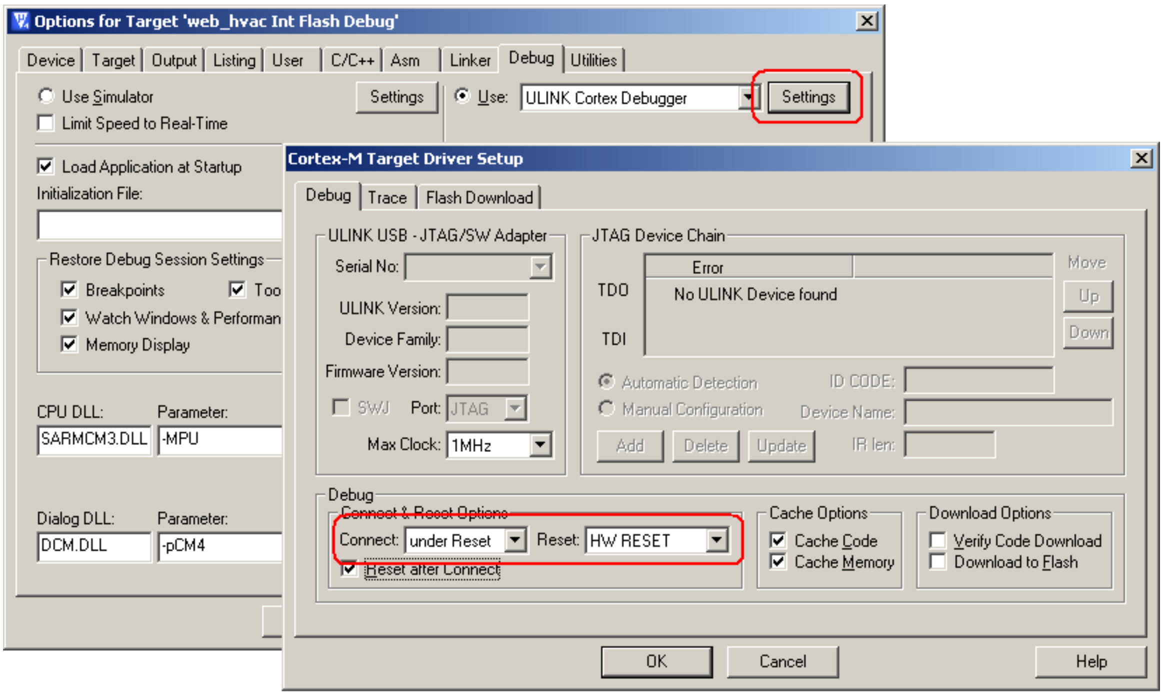
When the application is loaded, it is executed under the debugger and stops at initial breakpoint:


Disassembly
47: { /* Body */
48:
49:     extern const MQX_INITIALIZATION_STRUCT MQX_init_struct;
50:
51:     /* Start MQX */
0x00030FEC B510   PUSH   {r4,lr}
52:     _mqx((MQX_INITIALIZATION_STRUCT_PTR) &MQX_init_struct);
cortex_boot.s
mqx_main.c
40 *   Starts MQX running
41 *
42 *END*--*/
43 int main
44   (
45   void
46   )
47 { /* Body */
48
49   extern const MQX_INITIALIZATION_STRUCT MQX_init_struct;
50
51   /* Start MQX */
52   _mqx((MQX_INITIALIZATION_STRUCT_PTR) &MQX_init_struct);
53   return 0;
54
55 } /* Endbody */
56

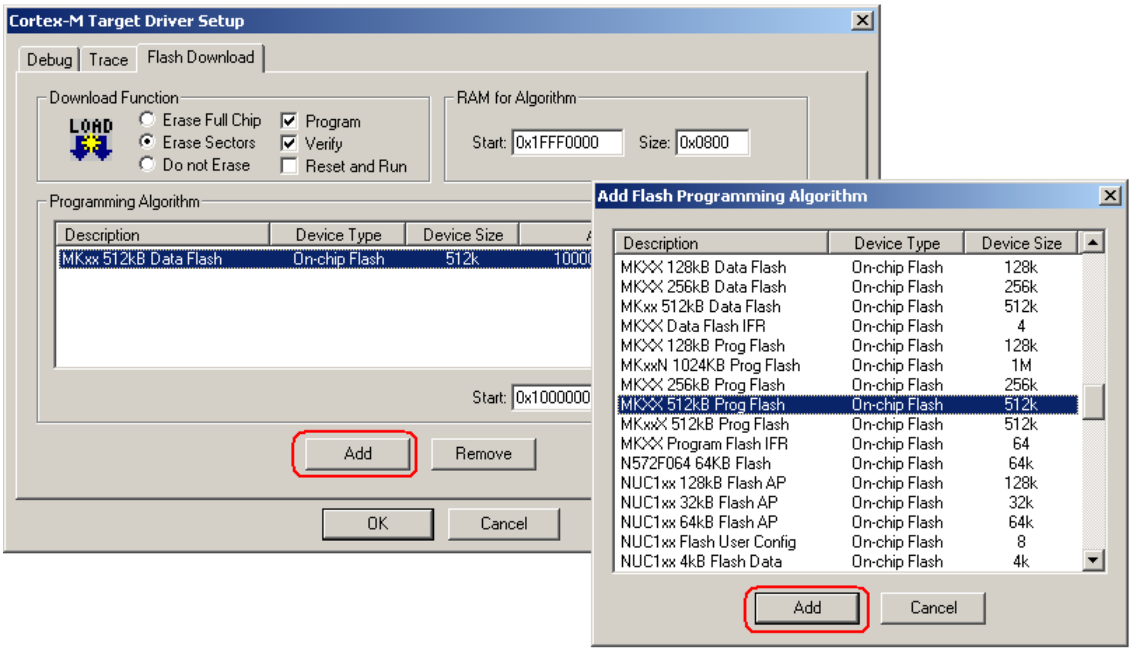
3.1.2 Using ARM Keil ULINK® Debugger Connection

Use of ULINK® or ULINK*pro* debugger interface requires additional configuration steps as described below.

In the project options, the “Debug” tab, select the ULINK connection, press the Settings button and make sure that the Connection options are set as shown in the image below:



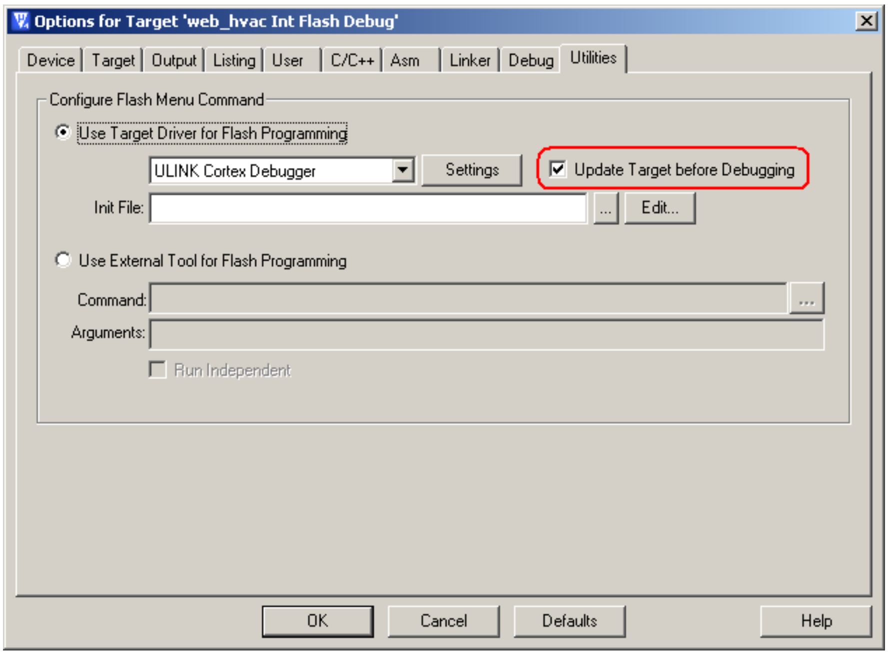
In the same setup dialog, select the “Flash Download” tab and add programming algorithm suitable for the target processor:



Select the Flash algorithm according to the following table:

Description	*Target processor*
MKXX 256kB Prog Flash	TWR-K40X256 Program Flash (0x00000000 – 0x00003FFFF)
MKXX 256kB Data Flash	TWR-K40X256 Data Flash (0x10000000 – 0x10003FFFF)
MKXX 512kB Prog Flash	TWR-K60N512 Program Flash (0x00000000 – 0x00007FFFF)
MKxxN 1024KB Prog Flash	TWR-K70FN1M Program Flash (0x00000000 – 0x0000FFFFF)

Back in the Options dialog, select the “Utilities” tab and make sure that the “Update Target before Debugging” option is selected:

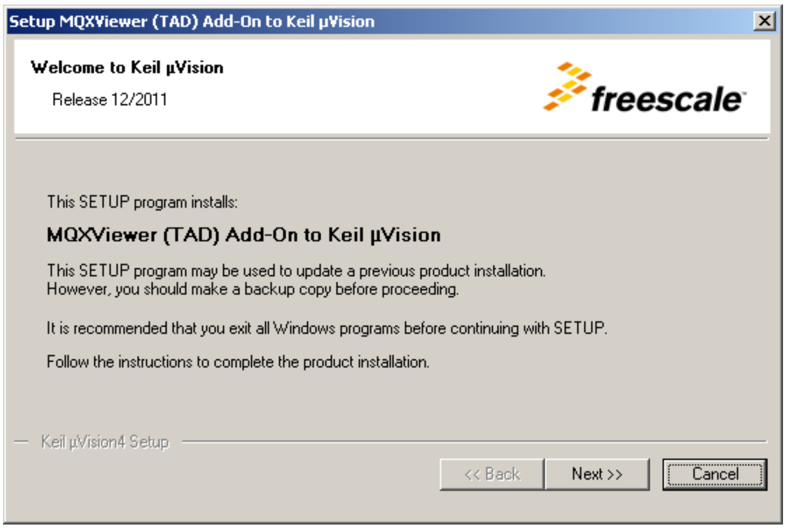


With the settings selected as described above, press the “Start/Stop Debug Session...” button to initiate the debugger session. It automatically loads the application to the target Flash memory and executes the application until it reaches initial breakpoint.

3.2 MQX-Viewer TAD Debugger Plug-in

3.2.1 Installing µVision4 TAD

The MQX-Viewer TAD plug-in is installed with add-on installer distributed inside the MQX installation package. The add-on installer is executed automatically during the MQX installation if µVision4 tool exists and if the Keil extensions are selected in MQX setup.

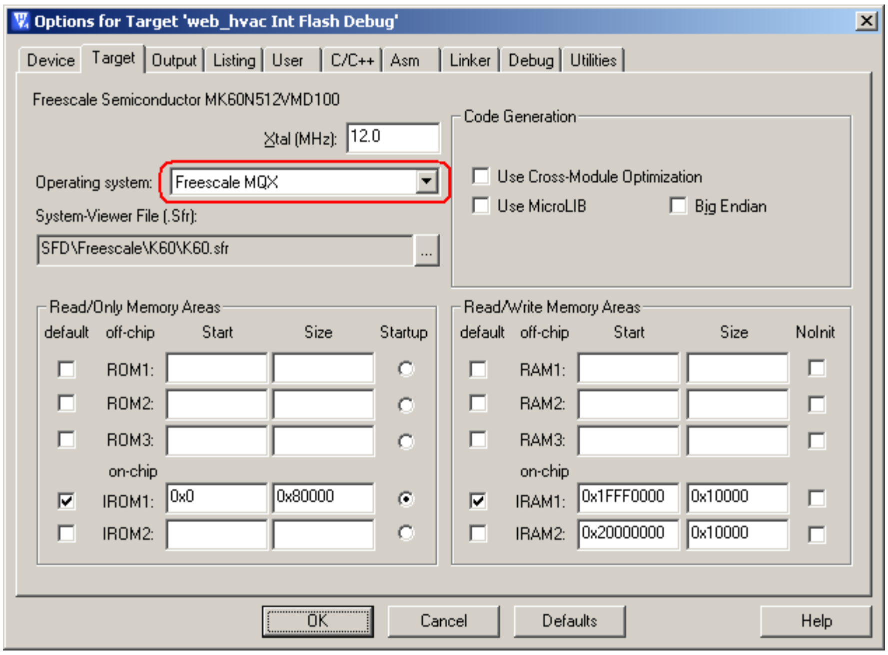


If you need to install the MQX-Viewer TAD plug-in manually after the MQX has been installed, simply start the add-on installer by running the following:

`<MQX Installation>\tools\keil_extensions\uVision4\MDK_MQX-Viewer_AddOn.exe`

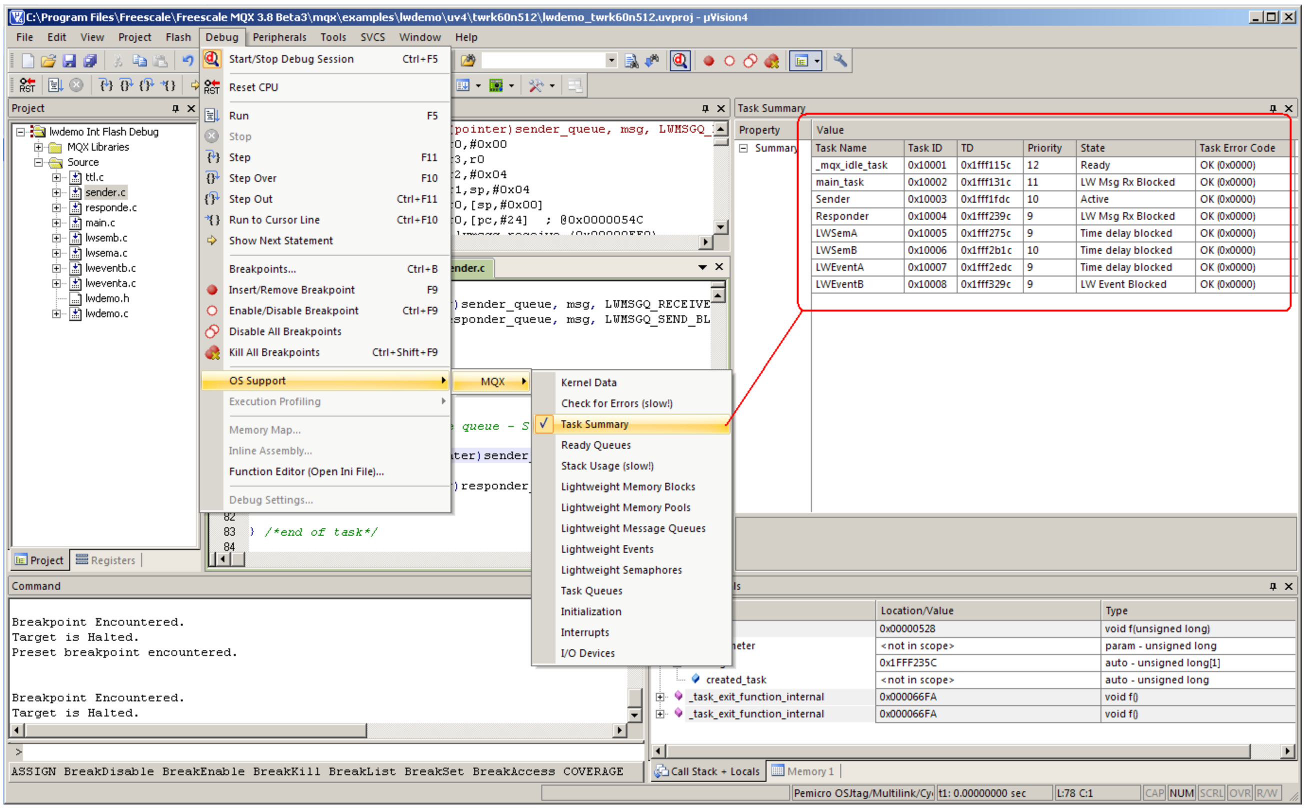
The add-on installer installs the plug-in as the `<uVision4>\ARM\BIN\MQX-Viewer.dll` file and registers it in the main TOOLS.ini file as `RTOSx=MQX-Viewer.dll ("Freescale MQX")` in the `[ARM]` and `[ARMADS]` sections.

When MQX-Viewer is installed, the MQX application projects should be set to use this plug-in during debugger sessions. In project options, select the “Target” tab and select the “Freescale MQX” operating system:



3.2.2 Using MQX-Viewer TAD Screens

Several TAD “screens” may be opened during the debugging session when using the MQX or RTCS menu in the µVision “OS Support” menu.



The most helpful and frequently used screens are shown in the images below:

Task Summary – overview of all tasks created in the MQX application.

Task Summary

Task Name	Task ID	TD	Priority	State	Task Error Code
_mqx_idle_task	0x10001	0x1fff115c	12	Ready	OK (0x0000)
main_task	0x10002	0x1fff131c	11	LW Msg Rx Blocked	OK (0x0000)
Sender	0x10003	0x1fff1fdc	10	Active	OK (0x0000)
Responder	0x10004	0x1fff239c	9	LW Msg Rx Blocked	OK (0x0000)
LWSemA	0x10005	0x1fff275c	9	Time delay blocked	OK (0x0000)
LWSemB	0x10006	0x1fff2b1c	10	Time delay blocked	OK (0x0000)
LWEventA	0x10007	0x1fff2edc	9	Time delay blocked	OK (0x0000)
LWEventB	0x10008	0x1fff329c	9	LW Event Blocked	OK (0x0000)

Stack Usage Summary – displays information about interrupt and task stacks. Typically, stack overflow is a root cause for the vast majority of problems in MQX user applications.

Stack Usage (slow!)

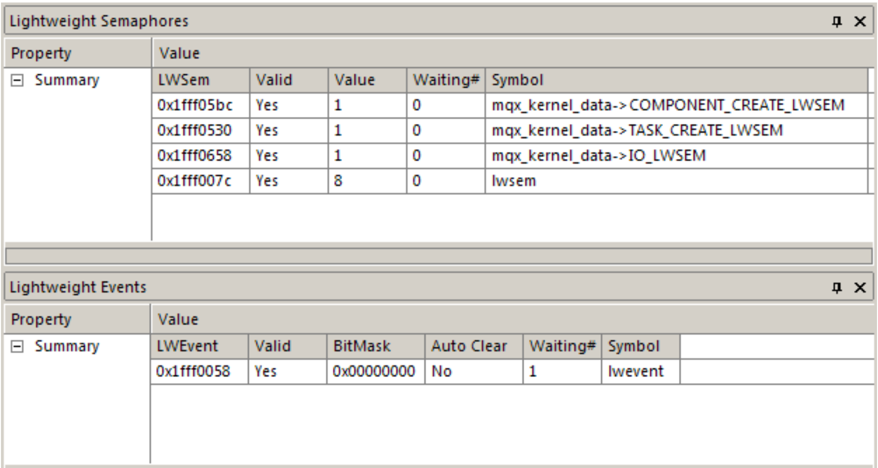
Property	Value					
⊟ Summary	Task	Stack Base	Stack Limit	Stack Used	% Used	Overflow?
	_mqx_idle_task	0x1fff1300	0x1fff1230	0x1fff129c	48 %	No
	main_task	0x1fff1fc0	0x1fff13d0	0x1fff1e5c	11 %	No
	Sender	0x1fff2380	0x1fff2090	0x1fff221c	47 %	No
	Responder	0x1fff2740	0x1fff2450	0x1fff2694	22 %	No
	LWSemA	0x1fff2b00	0x1fff2810	0x1fff2a6c	19 %	No
	LWSemB	0x1fff2ec0	0x1fff2bd0	0x1fff2e2c	19 %	No
	LWEventA	0x1fff3280	0x1fff2f90	0x1fff31ec	19 %	No
	LWEventB	0x1fff3640	0x1fff3350	0x1fff358c	23 %	No
	Interrupt	0x1fff0b60	0x1fff0760	0x1fff0b18	7 %	No

Lightweight Memory Block Summary – displays address, size, and type information about each memory block allocated in the default memory pool by the MQX system or applications. Additional memory pools, if used, may be displayed using the “Lightweight *Memory Pools*” screen.

Lightweight Memory Blocks

Property	Value				
Start:	0x1fff0750				
End:	0x2000fff0				
Size:	0x0001f8a0 (126.0K)				
Highest:	0x1fff3340 (10.0K, 8%)				
Pool valid?	Yes				
⊟ Summary	Address	Size (hex)	Size (dec)	Owner	Type
	0x1fff0750	0x420	1056	System	Interrupt Stack
	0x1fff0b70	0xf0	240	System	System Stack
	0x1fff0c60	0xe0	224	System	Ready Qs
	0x1fff0d40	0x90	144	System	Interrupt Table
	0x1fff0dd0	0x20	32	System	Interrupt Vector
	0x1fff0df0	0x20	32	System	Interrupt Vector
	0x1fff0e10	0x40	64	System	I/O Serial polled device struct
	0x1fff0e50	0x40	64	System	I/O Device
	0x1fff0e90	0x40	64	System	I/O Serial polled device struct
	0x1fff0ed0	0x40	64	System	I/O Device
	0x1fff0f10	0x30	48	System	I/O I2C polled device struct
	0x1fff0f40	0x40	64	System	I/O Device
	0x1fff0f80	0x30	48	System	I/O I2C int. device struct
	0x1fff0fb0	0x40	64	System	I/O Device
	0x1fff0ff0	0x30	48	System	
	0x1fff1020	0x40	64	System	I/O Device
	0x1fff1060	0x30	48	System	File
	0x1fff1090	0x60	96	System	I/O Serial charq
	0x1fff10f0	0x60	96	System	
	0x1fff1150	0xb0	176	0x10001	Task Descriptor
	0x1fff1200	0x110	272	0x10001	Task Stack
	0x1fff1310	0xb0	176	0x10002	Task Descriptor
	0x1fff13c0	0xc10	3088	0x10002	Task Stack
	0x1fff1fd0	0xb0	176	0x10003	Task Descriptor
	0x1fff2080	0x310	784	0x10003	Task Stack
	0x1fff2390	0xb0	176	0x10004	Task Descriptor
	0x1fff2440	0x310	784	0x10004	Task Stack
	0x1fff2750	0xb0	176	0x10005	Task Descriptor
	0x1fff2800	0x310	784	0x10005	Task Stack

Lightweight Semaphores, Lightweight Events – displays address and status of synchronization objects created by the MQX system or application. When a synchronization object is allocated as a global or static variable in the system, as an array element, or as a structure member allocated as a global or static variable, the TAD plug-in also displays the symbolic name of the object.

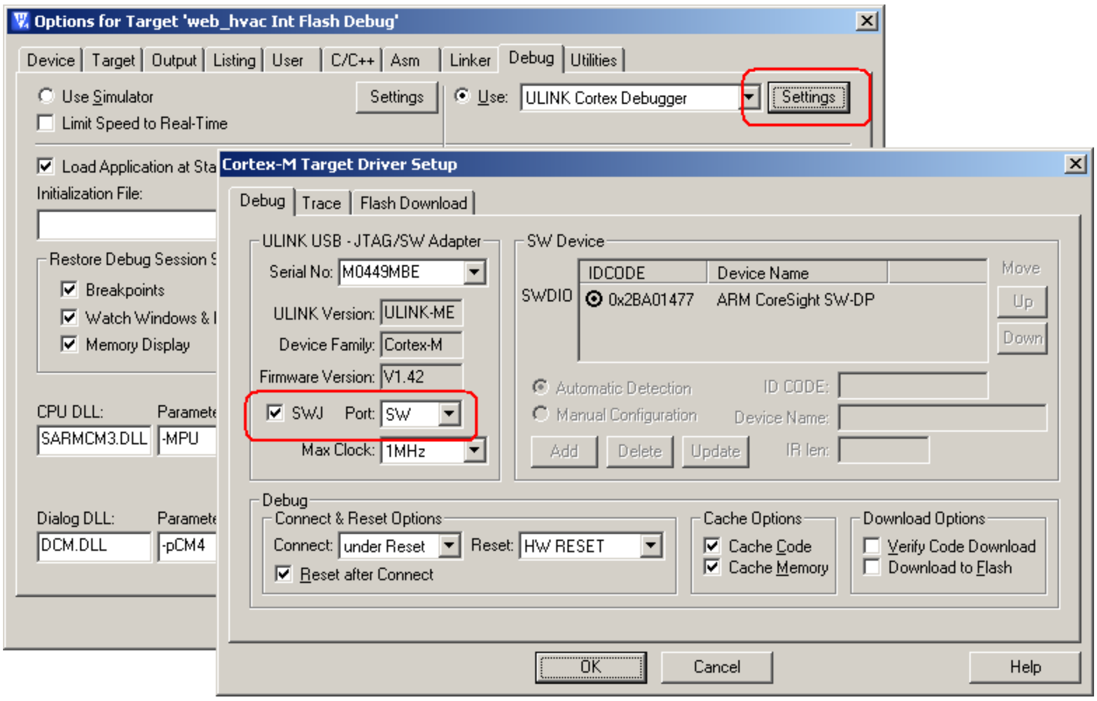


4 Using the MQX DebugI/O Driver with µVision4 IDE

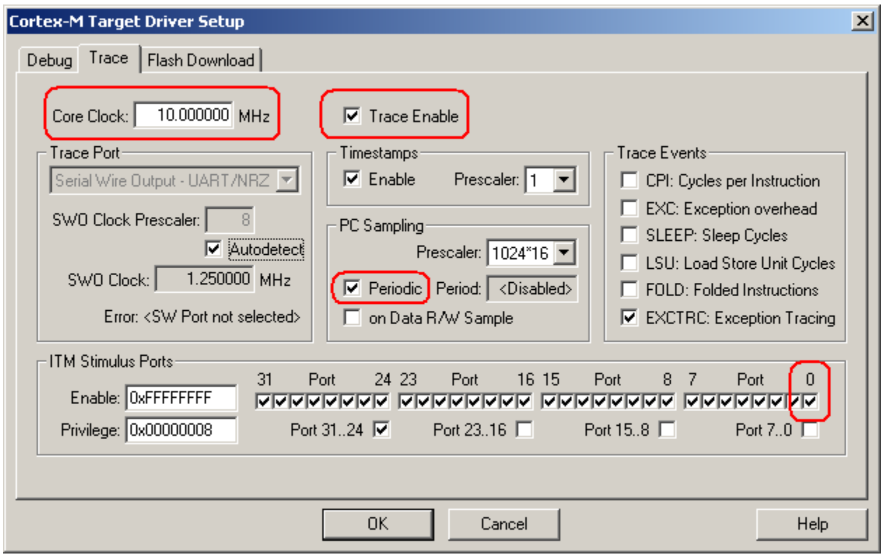
The MQX provides the DebugI/O driver allowing the processor to communicate with PC host computer via debugger probe. The DebugI/O channel can also be used as a default console for standard input and output operations. For more details about this driver, see *Getting Started with Freescale MQX™ RTOS* document.

The MQX RTOS currently supports ARM® Cortex®-M Semihost and ITM technologies. The µVision4® IDE supports the ITM communication channel over the ULINK® and ULINK*pro* debugger interfaces for both input and output directions.

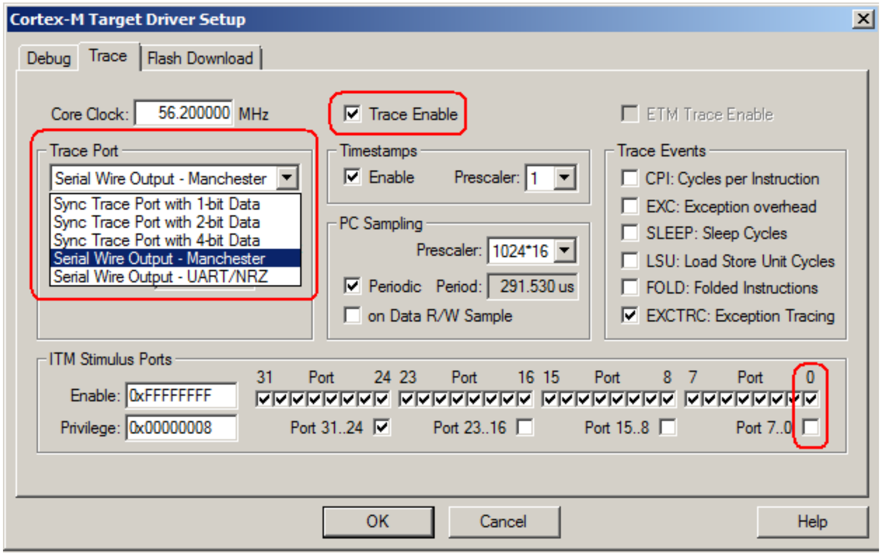
To set up the project for DebugI/O communication, open the project options at the "Debug" tab, select the ULINK connection, press the Settings button and make sure the SWJ and Port options are set as shown in the image below:



Switch to Trace tab and check “Trace enable”, ITM Port 0, and other settings as shown in the image below. The Core Clock should be set according to the real CPU setting:



For the ULINK*pro* debugger interface, select the Manchester coding of the Trace port as shown below:



The console window can be opened during a debug session using the “*View / Serial Windows / Debug (printf) Viewer”* menu.