Freescale Semiconductor
User's Guide

Document Number: MQX410IMX6SXGS
Rev. 0, 05/2015

Getting Started with Freescale MQX™ RTOS 4.1.0 on i.MX 6SoloX

1 Read Me First

This document describes how to configure the IAR Embedded Workbench®, ARM® Development Studio 5 (DS-5™), and Makefile development tool, as well as how to use them to build, run, and debug applications of the Freescale MQX™ RTOS. This document also provides board-specific information related to the MQX RTOS.

Contents

2 Building the MQX Libraries

2.1 Compile-time configuration

Major compile-time configuration options are grouped in a single user configuration file located in the following path:

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

This user configuration file is included internally by the private configuration files in MQX PSP and BSP and other core components, such as RTCS, USB, and Shell.

To share the configuration settings between different boards, add header files to the user_config.h file with common settings. The header files are located either in the same *<board>* directory or in the “common” directory:

<install_dir>/config/common

NOTE

Prior to MQX 4.0, the configuration header files were added from different locations when compiling the BSP/PSP and other library projects. The BSP and PSP code included the user_config.h file from the original /config folder, while other libraries, such as RTCS or MFS, were using a “copied” version of the this file from the /lib folder. This was changed in MQX 4.0 so that all libraries are using the user_config.h file from the /config folder.

This change removes the requirement to build the libraries in a certain order. Prior to MQX 4.0, the PSP and BSP libraries had to be built before the other libraries. This is no longer required.

2.2 Build process

After either the compile-time user configuration file or MQX kernel source files are changed, re-build the MQX libraries. The build process is similar for all the core components:

- The output directory is <install_dir>/lib/<board>.<compiler>/<target>/<component>, where <compiler> is a name of a build tool.
  For example, when the IAR tool is used to build MQX PSP and BSP libraries for i.MX 6SoloX SABRE-SDB board in debug targets, the libraries are built in the /lib/imx6sx_sdb_m4.iar/debug/psp and /lib/imx6sx_sdb_m4.iar/debug/bsp directories.

NOTE

In MQX 4.0, the library output paths were changed. The target names are now part of a library output path, while the name of the library file is the same for all targets. The Debug versions of the libraries are no longer created with the _d suffix, which makes it easier to create custom build targets and change different versions of libraries easily in the application projects.

- All public header files, needed for the application, are automatically copied from internal include folders to the same output directory as the library itself.
- During PSP or BSP build process, the user_config.h file and other header files from the config/<board> and config/common path are also copied into the lib/<board>.<compiler>/<target> output directory.
- After changing the file in /config/common/user_config.h, recompile all the MQX libraries.

CAUTION

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

2.3 Build targets

2.3.1 Library build targets

Each build project in the 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. Libraries are compiled into /lib/<board>.<compiler>/debug/<component>.
- **Release**: The compiler optimizations are set to the maximum. The compiled code is very hard to debug and should be used for final applications only. Libraries are compiled into /lib/<board>.<compiler>/release/<component>.

NOTE

The library path name pattern was changed in MQX 4.0. The _d suffix for debug was removed from the library. This change simplifies creating custom build targets and also enables changing different versions of libraries easily in the application projects. In order for the application to use libraries compiled in a different build target, set the proper library search paths in the application project settings. The library names, referred by the application project, remain unchanged. Different ABI options are no longer supported by the MQX RTOS. The user may add ABI builds as custom targets.

2.3.2 Application build targets

Build target names of MQX application projects reference either **Debug** or **Release** builds of the core libraries. Additionally, the target names also specify the built board memory configuration. For example:

- Ext Flash Debug/Release:
    - QSPI Flash: This target is suitable for final application deployment. When programmed to the QSPI Flash, the application starts immediately after the core is kicked off. Variables are allocated in the internal TCM memory. This is the default link target.
- RAM Debug/Release:
    - OCRAM: These targets are for MQX RTOS application debug only, and should not run at the same time as the Linux® OS to avoid memory conflicts. The application will first be loaded to OCRAM by bootloader and then be kicked off. Variables are allocated in the internal TCM memory. To build with RAM target, you have to modify the link file setting in the build project.

2.3.3 Settings changes to switch to the RAM target

- For IAR Embedded Workbench or ARM DS-5

  Change the link file to ram.xxx in project. For example, change from lib/imx6sx_sdb_m4.iar/debug/bsp/extflash.icf to lib/imx6sx_sdb_m4.iar/debug/bsp/ram.icf in the red boxes in Figure 1 and Figure 2.

- For Makefile

  Edit <install_dir>/build/common/make/global.mak.

    - Find GET_BSP_LINKER_FILE = $(firstword $(wildcard …)).
    - Set GET_BSP_LINKER_FILE=<RAM link file path>.

    Once it is complete, all applications built with GCC will change to the RAM target.

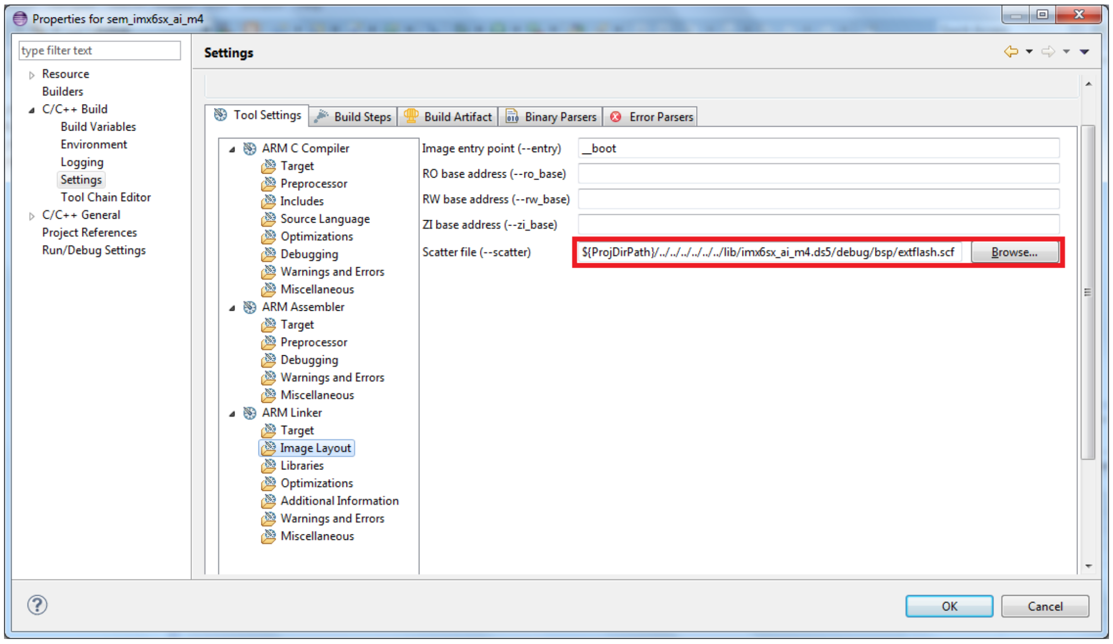
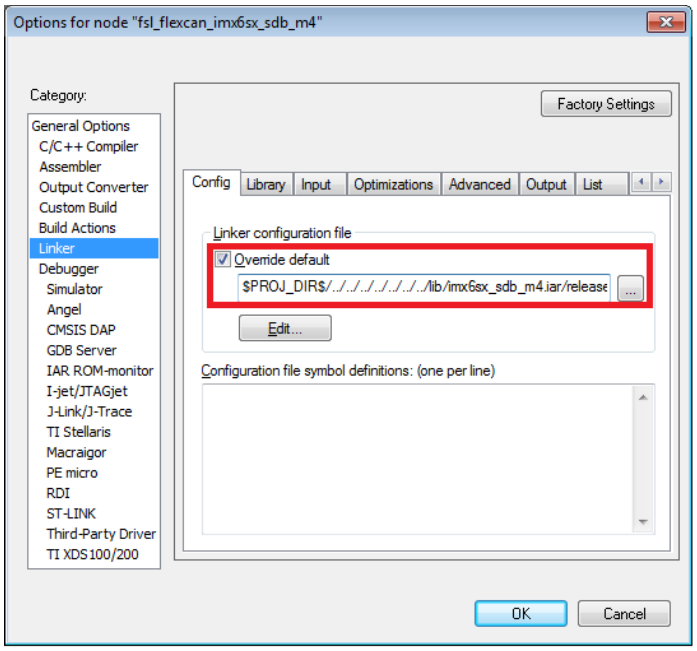


Image entry point (--entry) __boot
RO base address (--ro_base)
RW base address (--rw_base)
ZI base address (--zi_base)
Scatter file (--scatter) ${ProjDirPath}/../../../../../../lib/imx6sx_ai_m4.ds5/debug/bsp/extflash.scf

Figure 1 Scatter file in DS5 project properties



Linker configuration file

[✓] Override default

$PROJ_DIR$/../../../../../../../lib/imx6sx_sdb_m4.iar/release

Figure 2 Linker configuration file in IAR project options

2.3.4 Compiler and Linker Warning settings

Several compiler and linker warnings are suppressed in the MQX release.

iar:

- Pa082: Undefined behavior. The order of volatile accesses is undefined in this statement.
- Pe186: Pointless comparison of unsigned integer with zero.
- Pe17: Variable “[name]” was declared but never referenced.
- Pe550: Variable “[name]” was set but never used.
- Pe174: Expression has no effect.

ds5:

- 1609: The instruction does not set bit zero, so it does not create a return address.
- 186: Pointless comparison of unsigned integer with zero.
- 1296: Extended constant initializer used.

gcc:

- -Wno-missing-braces - bug in gcc
- -Wno-switch

- -Wno-unused-value
- -Wno-unused-variable
- -Wno-unused-but-set-variable
- -Wno-pointer-to-int-cast
- -Wno-unused-function
- -Wno-unused-label
- -Wno-char-subscripts
- -Wno-int-to-pointer-cast

2.4 IAR embedded workbench for ARM

To rebuild the MQX libraries, open and batch-build this IAR EWARM workspace:

<mqx_install_dir>/build/<board>/iar/build_libs.eww

The output directory is as follows:

<mqx_install_dir>/lib/<board>.iar/

For detailed information about the MQX support in IAR tools, see *Getting Started with Freescale MQX™ RTOS and IAR Embedded Workbench®* (document MQXGSIAR). The document is included in the MQX installation in <mqx_install_dir>/doc/tools/iar/MQX-IAR-Getting-Started.pdf.

2.5 ARM Development Studio 5 (DS-5™) environment support

First, install the MQX Eclipse plug-in by using Help\Install New Software\Add\Archive… menu.

Then, select this archive:

<mqx_install_dir>/tools/ds5/ds5_update_site.zip

To rebuild the MQX libraries, first import the <board>.wsd working set description file by using the "File\Import\MQX\Import Working Sets" menu. The MQX library projects will be imported to the DS-5 working space together with build configurations settings.

The location of working set description (wsd) file is as follows:

<mqx_install_dir>/build/<board>/ds5/<board>.wsd

The output directory is as follows:

<mqx_install_dir>/lib/<board>.ds5

For detailed information about the MQX support in ARM tools, see *Getting Started with ARM® Design Studio 5 (DS-5™) with Freescale MQX™ RTOS* (document MQXGSDS5). The document is included in the MQX installation in <mqx_install_dir>/doc/tools/ds5/MQX-DS5-Getting-Started.pdf.

2.6 Building MQX RTOS using makefiles

The MQX version 4.1 contains the Makefiles to build libraries and applications for selected platforms by using the command-line.

The support for Makefile command line builds will be expanded to cover all boards and BSPs in the later MQX versions.

Use the mingw32-make version 3.8.2 or higher on Microsoft Windows operating system. Download the latest version from sourceforge.net/projects/mingw/.

To rebuild the MQX libraries navigate to <mqx_install_dir>/build/<board>/make and run this command (building MQX RTOS using the GCC ARM of the toolchain):

C:\MinGW\bin\mingw32-make.exe TOOL=gcc_arm CONFIG=debug build

NOTE

Prior to the build, specify the path (*`TOOLCHAIN_ROOTDIR`* variable) for your build tools in <mqx_install_dir>/build/common/make/global.mak.

3 Porting User Applications to MQX RTOS 4.1.0

3.1 C99 types

Starting with the MQX version 4.1.0, the standard C99 integer and boolean types (from **stdlib.h** and **stdbool.h**) are used instead of the proprietary MQX types. When porting the application to the MQX version 4.1.0, compilation issues may occur.

The **psptypes_legacy.h** header file is available and contains the definition of the MQX legacy types. This file can be included in the MQX application to overcome most of the compilation problems caused by the change of types.

To avoid potential problems, use the C99 types in your application.

3.2 MQX RTOS startup split

To avoid a crash risk if an interrupt occurs during the startup, the MQX RTOS startup is split in two parts.

- The _bsp_enable_card() function has been replaced by the _bsp_pre_init() function that handles initialization of the OS vital functions, such as the timer (system tick), interrupt controller, memory management, etc. The _bsp_pre_init() function is called in the MQX RTOS initialization time, before any user task is created and the scheduler is not started.
- The second part of the startup is done in a separate _mqx_init_task that executes _bsp_init() function for I/O drivers or stacks initialization and _bsp_post_init() function for possible post-init operations. After the _bsp_post_init() function execution, the _mqx_init_task is destroyed.

All BSPs are adjusted to this concept. All I/O drivers are installed in the context of the _mqx_init_task after the MQX scheduler is started. This concept also allows a complex driver installation (handling ISRs during the driver initialization, drivers could use blocking functionality like _time_delay, etc.).

When porting your BSP to the MQX RTOS 4.1.0, split the _bsp_enable_card() function (init_bsp.c) into the_bsp_pre_init() and the _bsp_init() functions. All _bsp_enable_card() code up to the I/O subsystem initialization are part of the new _bsp_pre_init() function. The rest of the _bsp_enable_card() code is part of the new _bsp_init() function.

4 Creating a New MQX RTOS Project

Typically, the process of creating new projects mostly depends on the development environment being used. Describing this process is out of scope of this document. For more information, see the documentation provided by the tool vendor.

A general recommendation for starting a new MQX RTOS project in any IDE environment is to clone one of the existing example applications, save it under a custom name, and modify it to meet your specific needs. In this situation, be aware that there may be relative paths to support files referred to in the project. This may apply to search paths, linker command files, debugger, configuration files, etc. Make sure that you update the relative paths in the new “clone” of the project.

5 MQX RTOS Board Support Package Configuration Options

5.1 Standard input and output channel settings

An I/O communication device installed by MQX RTOS BSP can be used as the standard I/O channel, such as Serial line. The default console settings for each supported development board are specified in Section 7 "Board-specific Information Related to MQX".

Freescale boards provide several ways to connect the I/O console. The most common options are as follows:

- Serial I/O is a channel routed through the RS232 connector. This port can be connected directly to the PC serial interface and used with a suitable terminal program (e.g., Hyper-Terminal).
- Serial I/O is a channel routed through combined OSBDM/OSJTAG debugging and communication port directly to the board. For microcontrollers, the communication interface is connected to a serial (SCI/UART) port. For a PC-Host, you can use either a virtual USB serial port driver or a special USB communication terminal application. See the development board documentation for more details.

The default serial I/O Channel Settings are as follows:

Baud Rate 115200

- Data bits 8
- No parity
- 1 stop bit and no flow control

Pre-defined default I/O channel setting is specified for each board in the header file in `mqx\source\bsp\<board_name>\board_name.h`. This setting can be overridden in the user_config.h file by adding this code and rebuilding the BSP library.

To set the default I/O channel to the UART2 serial interface (mapped to `ittyb:` device in MQX RTOS), use this:

```
#define BSP_DEFAULT_IO_CHANNEL           "ittyb:"
```

Ensure that the serial channel (ttyb: in this instance) is in enabled in `user_config.h` file as follows:

```
#define BSPCFG_ENABLE_ITTYB                1
```

6 Running the Examples

This section describes how to run the MQX RTOS application by itself without Linux OS running on the ARM® Cortex®-A9 processor. To run the Linux OS and MQX RTOS at the same time on the i.MX 6SoloX SABRE-SDB or SABRE-AI board, load the MQX RTOS with the U-Boot loader in the Linux package, using the same steps in the following section.

6.1 Running with U-Boot

There is a prebuilt U-Boot image available to kick off the MQX RTOS application. You can find it at <install_dir>/tools/u-boot-<board>.imx. This is the U-Boot loader for running MQX RTOS only.

The default boot configuration of the SDB board is boot from SD4, and on the SABRE-AI board, there's only one SD slot that is used for boot. You need to prepare an SD card, and then create a single FAT partition with 2 MB offset from MBR (by Microsoft Windows® 7 "diskpart" tool, or Ubuntu "fdisk" tool), and then write U-Boot at 1 KB offset (by Windows application ddcopy.exe located in <install_dir>/tools/ddcopy, or Ubuntu "dd" tool).

This section only shows how to do it on Microsoft Windows 7.

1. Open "command prompt", and run "diskpart".
    a. `list disk`: Shows all the available disks.
    b. `select disk [num]`: Choose the disk number that represents your SD card, such as 1.
    c. `clean all`: Remove all the partitions from disk 1, which takes some time to complete.
    d. `create partition primary offset=2048 size=512`: Create the primary partition of 512 MB with the offset of 2048 KB.
    e. Safely remove the SD card from the slot and plug it in again. Then you can format the partition to the FAT file system.
2. Then you can write U-Boot to the SD card, insert your SD card, and get the Windows partition number, such as "F:".Open "command prompt", and run "cmd".
    a. Change the directory to <install_dir>/tools/ddcopy.
    b. Run `ddcopy.exe infile=../u-boot-<board>.imx outdevice=F: seek=0x400 obs=512`.
    c. Safely remove the SD card from the slot.

    You can now prepare the MQX RTOS application image to be loaded by U-Boot. For instructions on how to build applications to create a binary image, see the getting started guides of your toolchain. After the binary image is ready, you can copy it to the FAT system on the SD card.
3. Insert the SD card to the boot slot, and connect the "UART to USB" slot on the board with your PC through the USB cable. The Windows OS will install the USB driver automatically, and Ubuntu OS will find the serial devices as well.
    - On the Windows OS, open the device manager, find "USB serial Port" in "Ports (COM and LPT)". Assume that the ports are COM7 and COM8. The smaller numbered port (COM7)

is for the debugging message from ARM Cortex-A9 processor and the larger numbered port (COM8) is for ARM® Cortex®-M4 processor.

- On the Ubuntu OS, find the TTY device with name /dev/ttyUSB* to determine your debugging port. Similar to the Windows OS, the smaller number is for ARM Cortex-A9 processor and the bigger number is for ARM Cortex-M4 processor.

Open your favorite serial terminals for the serial devices, set the speed to 115200 bps, data bits 8, no parity, and power on the board.

4. On the COM7 terminal, press any key within 3 seconds of booting, and U-Boot will enter command line mode, and then you can write your MQX RTOS application to the QSPI flash.

   For the i.MX 6SoloX SABRE-SDB board, the U-Boot commands are as follows:

   a. `fatload mmc 2:1 0x80800000 image.bin`: Load the flash image file from the SD card to DDR.
   b. `sf probe 1:0`: Load the SPI flash driver.
   c. You should get the message "`SF: Detected N25Q256 with page size 256 Bytes, erase size 4 KiB, total 32 MiB`".
   d. `sf erase 0x0 0x40000`: Erase the first 256 KB in the flash.
   e. `sf write 0x80800000 0x0 0x40000`: Burn the image from DDR to the flash.
   f. `bootaux 0x78000000`: Start the Cortex-M4 processor at the flash head.

   For the i.MX 6SoloX SABRE-AI board, the U-Boot commands are slightly different:

   a. `fatload mmc 0:1 0x80800000 image.bin`: Load the flash image file from the SD card to DDR.
   b. `sf probe 1:0`: Load the SPI flash driver.
   c. You should get the message "`SF: Detected N25Q256 with page size 256 Bytes, erase size 4 KiB, total 32 MiB`".
   d. `sf erase 0x0 0x40000`: Erase the first 256 KB in the flash.
   e. `sf write 0x80800000 0x0 0x40000`: Burn the image from DDR to the flash.
   f. `bootaux 0x68000000`: Start the Cortex-M4 processor at the flash head.

   The MQX RTOS application message is displayed on the COM8 terminal. Because the image is on the flash, if you want to run the same image again, reboot the board and repeat Step b "`sf probe 1:0`"and Step f "`bootaux 0x78000000`" for the SABRE-SDB board, or "`bootaux 0x68000000`" for SABRE-AI board and you can also add these commands to the "bootcmd" variable of U-Boot to run it automatically.

6.2 Running with the TRACE32 debugger (on the Windows OS only)

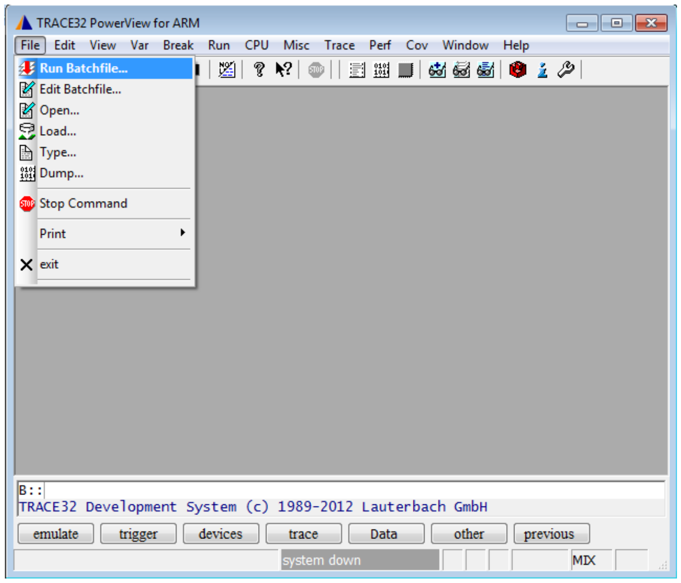
To debug the program with TRACE32, a script file is used. Find it at <install_dir>/tools/trace32/attach_imx6sx_m4.cmm.

Make sure that TRACE32 ICD (In-Circuit-Debugger) for ARM is installed, and your Lauterbach debugger device supports Cortex-M4 debugging.

Build an MQX RTOS application of the **RAM** target, and change the default ELF load path in `attach_imx6sx_m4.cmm`.

data.load.elf "<your ELF path>" /verify

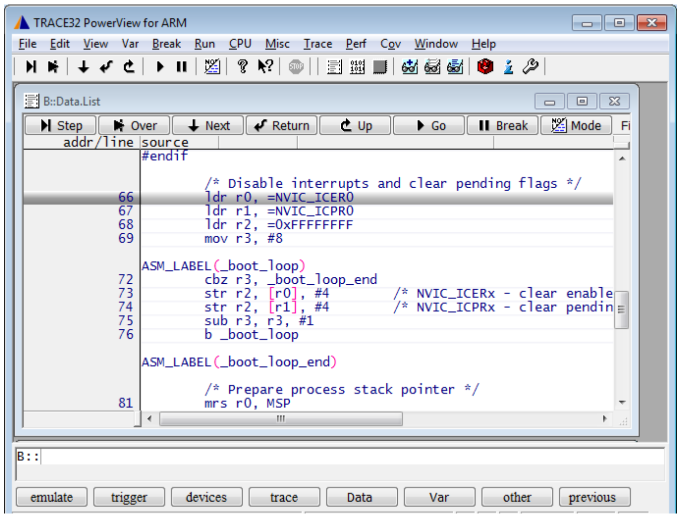
Connect the TRACE32 debugger device to your PC and the board (through JTAG). Run the TRACE32 ICD ARM debugger, and load `attach_imx6sx_m4.cmm` by choosing **File -> Run Batchfile**.


TRACE32 PowerView for ARM
File Edit View Var Break Run CPU Misc Trace Perf Cov Window Help
Run Batchfile...
Edit Batchfile...
Open...
Load...
Type...
Dump...
Stop Command
Print
exit
emulate trigger devices trace Data other previous
system down
MIX

```
B::
TRACE32 Development System (c) 1989-2012 Lauterbach GmbH
```

Figure 3 Running the script file

Now you can run (GO) and debug the program with the single step (Step, Over, Next, Return) or break points.


TRACE32 PowerView for ARM
File Edit View Var Break Run CPU Misc Trace Perf Cov Window Help
B::Data.List
Step Over Next Return Up Go Break Mode Fi
addr/line source
B::
emulate trigger devices trace Data Var other previous

```
#endif

        /* Disable interrupts and clear pending flags */
        ldr r0, =NVIC_ICER0
        ldr r1, =NVIC_ICPR0
        ldr r2, =0xFFFFFFFF
        mov r3, #8

ASM_LABEL(_boot_loop)
        cbz r3, _boot_loop_end
        str r2, [r0], #4        /* NVIC_ICERx - clear enable
        str r2, [r1], #4        /* NVIC_ICPRx - clear pendin
        sub r3, r3, #1
        b _boot_loop

ASM_LABEL(_boot_loop_end)

        /* Prepare process stack pointer */
        mrs r0, MSP
```

Figure 4 Starting debugging

For more information about the TRACE32 debugger, see icd_tutorial.pdf in your installation directory of TRACE32, such as. C:/T32/pdf.

6.3 Task aware debugging plug-in

MQX RTOS Task Aware Debugging plug-in (TAD) is an optional extension to a debugger tool that enables easy debugging for multi-task applications. It helps to visualize internal MQX RTOS data structures, task-specific information, I/O device drivers, and other MQX RTOS context data.

The MQX TAD is supported by Lauterbach (TAD plug-in available directly from Lauterbach). You can find the manual in your TRACE32 installation folder, such as C:/T32/pdf/rtos_mqx.pdf.

NOTE

The TAD plug-in communicates with the debugger and works with MQX RTOS data obtained directly from the RAM memory. A typical debugger session is configured in a way that the code execution stops at the first breakpoint in the *main()* function. In this function, MQX RTOS internal data structures are not yet fully initialized and the TAD functionality is significantly reduced. All TAD features are available only when execution stops at a breakpoint in the application task code.

7 Board-Specific Information Related to MQX RTOS

This section provides more details about all boards and BSPs supported by the current MQX RTOS distribution.

All jumper and other hardware switches, not specifically described below, are assumed to be in factory-default positions. See board user guides for default settings.

7.1 i.MX 6SoloX

7.1.1 i.MX 6SoloX SABRE-SDB

Figure 5 shows the i.MX 6SoloX SABRE-SDB board.

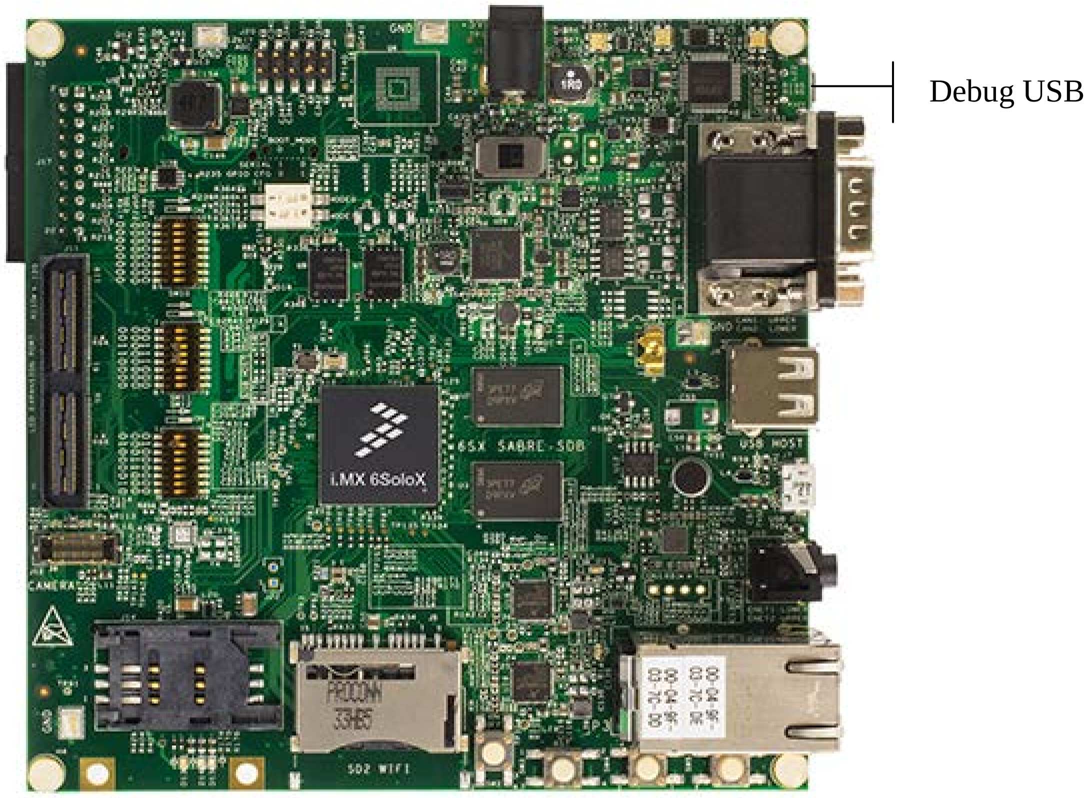


Figure 5 i.MX 6SoloX SABRE-SDB board

Table 1 ARM Cortex-M4 processor – imx6sx_sdb_m4 BSP

Core Clock	227 Mhz	
Bus Clock	132 Mhz	
Default Console	ittyb:	Debug USB
BSP Timer	SysTick	

The important jumper settings (Rev. B) are as follows:

- J18 fitted
- SW10: 00000000
- SW11: 00111000
- SW12: 01000000

The known issues are as follows:

In SPI master and slave example, it will reuse SD2 (WIFI) socket pins for connection, and this will conflict with the Linux OS running on Cortex-A9 processor. Make sure to only run the SPI master/slave example without the Linux OS.

7.1.2 i.MX 6SoloX SABRE-AI

Figure 6 shows the i.MX 6SoloX SABRE-AI board.

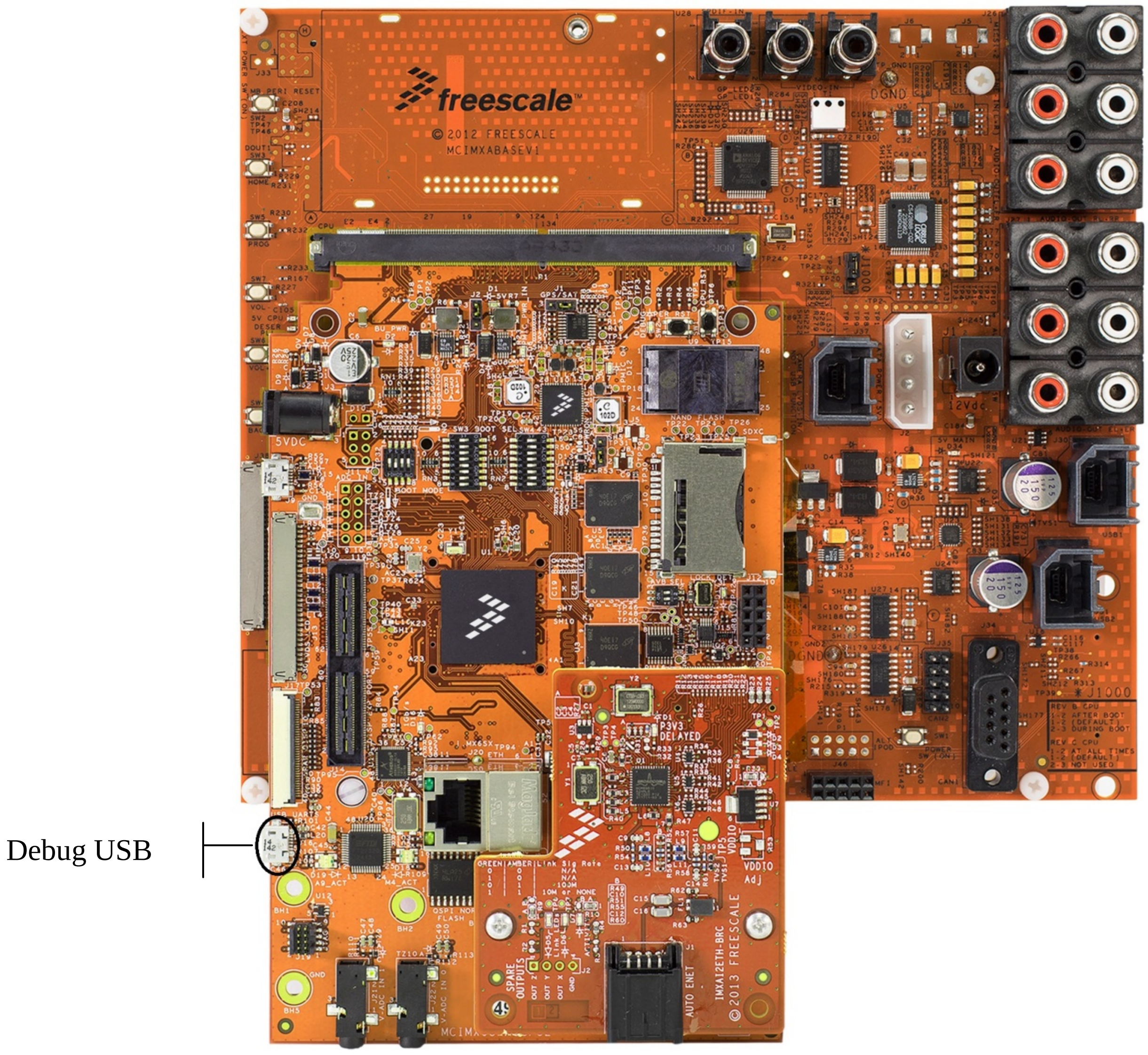


Figure 6 i.MX 6SoloX SABRE-AI board

Table 2 ARM Cortex-M4 processor – imx6sx_ai_m4 BSP

Core Clock	227 Mhz	
Bus Clock	132 Mhz	
Default Console	ittyb:	Debug USB
BSP Timer	SysTick	

The important jumper settings (Rev. A) are as follows:

- J1: 1-2 fitted
- J2:
    - 1-2 fitted without extension board
    - 2-3 fitted with extension board
- J3: 2-3 fitted
- SW3: 00001100
- SW4: 01000010
- BOOT MODE: 0010

8 Revision History

Table 3 Revision history

Revision number	Date	Substantive changes
0	05/2015	Initial release.

Document Number: MQX410IMX6SXGS
Rev. 0
05/2015