bootlin

Using the Cortex-M4 MCU on the i.MX6 SoloX from Linux

Introduction

The NXP i.MX6 SoloX System on Chip has two different CPU cores (i.e. Assymetric Multi Processing), a Cortex-A9 and a Cortex-M4. The Cortex-M4 MCU allows running an hard real-time OS while still having access to all the SoC peripherals.

This post is about running an application on the Cortex-M4, loading it from the Linux userspace. The i.MX 6SoloX SABRE Development Board is used for this demonstration.
It doesn't describe in details how to build the BSP but the meta-freescale Yocto Project layer has been used. The kernel is a vendor kernel derivative, linux-fslc.

Building the cortex M4 binary

We will be running FreeRTOS on the Cortex-M4. The BSP is available from the NXP website (it may require registration).

Uncompress it:

```
$ tar xf FreeRTOS_BSP_1.0.1_iMX6SX.tar.gz
```

A baremetal toolchain is needed to compile for Cortex-M4. One is available from ARM.

Download it and uncompress it:

```
$ wget https://armkeil.blob.core.windows.net/developer/Files/downloads/gnu-rm/7-2018q2/gcc-arm-none-eabi-7-2018-q2-update-linux.tar.bz2
$ tar xf gcc-arm-none-eabi-7-2018-q2-update-linux.tar.bz2
```

The provided examples will have the Cortex-M4 output debug on one of the SoC UART. Unfortunately, the hardware setup code will forcefully try to use the 24MHz oscillator as the clock parent for the UART and because Linux is using a 80MHz clock, running the examples as-is would result in a non functioning Linux console.

The following patch solves this issue:

```
diff -burp a/examples/imx6sx_sdb_m4/board.c b/examples/imx6sx_sdb_m4/board.c
--- a/examples/imx6sx_sdb_m4/board.c
+++ b/examples/imx6sx_sdb_m4/board.c
@@ -69,10 +69,12 @@ void dbg_uart_init(void)
     /* Set debug uart for M4 core domain access only */
     RDC_SetPdapAccess(RDC, BOARD_DEBUG_UART_RDC_PDAP, 3 << (BOARD_DOMAIN_ID * 2), false, false);

+#if 0
     /* Select board debug clock derived from OSC clock(24M) */
     CCM_SetRootMux(CCM, ccmRootUartClkSel, ccmRootmuxUartClkOsc24m);
     /* Set relevant divider = 1. */
     CCM_SetRootDivider(CCM, ccmRootUartClkPodf, 0);
+#endif
     /* Enable debug uart clock */
     CCM_ControlGate(CCM, ccmCcgrGateUartClk, ccmClockNeededAll);
     CCM_ControlGate(CCM, ccmCcgrGateUartSerialClk, ccmClockNeededAll);
@@ -80,7 +82,7 @@ void dbg_uart_init(void)
     /* Configure the pin IOMUX */
     configure_uart_pins(BOARD_DEBUG_UART_BASEADDR);

-    DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR, 24000000, 115200);
+    DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR, 80000000, 115200);
 }

 /*FUNCTION*----------------------------------------------------------------
```

After applying that, let's build some examples.

```
$ cd FreeRTOS_BSP_1.0.1_iMX6SX/examples/imx6sx_sdb_m4/demo_apps/hello_world/armgcc
$ ARMGCC_DIR=~/gcc-arm-none-eabi-7-2018-q2-update ./build_release.sh
...
[100%] Linking C executable release/hello_world.elf
[100%] Built target hello_world
$ cd ../../rpmsg/pingpong_freertos/armgcc/
$ ARMGCC_DIR=~/gcc-arm-none-eabi-7-2018-q2-update ./build_release.sh
...
[100%] Linking C executable release/rpmsg_pingpong_freertos_example.elf
[100%] Built target rpmsg_pingpong_freertos_example
$ cd ../../str_echo_freertos/armgcc/i
$ ARMGCC_DIR=~/gcc-arm-none-eabi-7-2018-q2-update ./build_release.sh
...
[100%] Linking C executable release/rpmsg_str_echo_freertos_example.elf
[100%] Built target rpmsg_str_echo_freertos_example
```

The generated binaries are in *FreeRTOS_BSP_1.0.1_iMX6SX/examples/imx6sx_sdb_m4/demo_apps/*:

- *rpmsg/pingpong_freertos/armgcc/release/rpmsg_pingpong_freertos_example.bin*
- *rpmsg/str_echo_freertos/armgcc/release/rpmsg_str_echo_freertos_example.bin*
- *hello_world/armgcc/release/hello_world.bin*

Copy them to your root filessystem.

Loading the M4 binary

NXP provides a userspace application allowing to load the binary on the Cortex-M4 from Linux: imx-m4fwloader.

Unfortunately, this doesn't work out of the box because the code in the vendor kernel is expecting the Cortex-M4 to be started by u-boot to initialize communication channels between the Cortex-A9 and the Cortex-M4, specifically the shared memory area containing the clocks status and the Cortex-M4 clock.

The following hack ensures the initialization is done and the Cortex-M4 clock is left enabled after boot:

```
diff --git a/arch/arm/mach-imx/src.c b/arch/arm/mach-imx/src.c
index c53b6da411b9..9859751b5109 100644
--- a/arch/arm/mach-imx/src.c
+++ b/arch/arm/mach-imx/src.c
@@ -194,6 +194,7 @@ void __init imx_src_init(void)
                m4_is_enabled = true;
        else
                m4_is_enabled = false;
+       m4_is_enabled = true;

        val &= ~(1 << BP_SRC_SCR_WARM_RESET_ENABLE);
        writel_relaxed(val, src_base + SRC_SCR);
```

Don't forget to use the *-m4* version of the device tree (i.e *imx6sx-sdb-m4.dtb*), else the kernel will crash with the following dump:

```
Unable to handle kernel NULL pointer dereference at virtual address 00000018
pgd = 80004000
[00000018] *pgd=00000000
Internal error: Oops: 805 [#1] PREEMPT SMP ARM
Modules linked in:
CPU: 0 PID: 1 Comm: swapper/0 Not tainted 4.9.67-fslc-02224-g953c6e30c970-dirty #6
Hardware name: Freescale i.MX6 SoloX (Device Tree)
task: a80bc000 task.stack: a8120000
PC is at imx_amp_power_init+0x98/0xdc
LR is at 0xa8003b00
pc : [<810462ec>]    lr : []    psr: 80000013
sp : a8121ec0  ip : 812241cc  fp : a8121ed4
r10: 8109083c  r9 : 00000008  r8 : 00000000
r7 : 81090838  r6 : 811c9000  r5 : ffffe000  r4 : 81223d78
r3 : 00000001  r2 : 0000001c  r1 : 00000001  r0 : 00000000
Flags: Nzcv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment none
Control: 10c53c7d  Table: 8000404a  DAC: 00000051
Process swapper/0 (pid: 1, stack limit = 0xa8120210)
Stack: (0xa8121ec0 to 0xa8122000)
1ec0: 81046254 ffffe000 a8121f4c a8121ed8 80101c7c 81046260 81000638 8045f190
1ee0: abfff9ad 80b35338 a8121f00 a8121ef8 801522e0 81000628 a8121f34 80d4aefc
1f00: 80d4a754 80d57578 00000007 00000007 00000000 80e59e78 80d96100 00000000
1f20: 81090818 80e59e78 811c9000 80e59e78 810c3c70 811c9000 81090838 811c9000
1f40: a8121f94 a8121f50 81000ea0 80101c34 00000007 00000007 00000000 8100061c
1f60: 8100061c 00000130 dc911044 00000000 80ab005c 00000000 00000000 00000000
1f80: 00000000 00000000 a8121fac a8121f98 80ab0074 81000d40 00000000 80ab005c
1fa0: 00000000 a8121fb0 80108378 80ab0068 00000000 00000000 00000000 00000000
1fc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1fe0: 00000000 00000000 00000000 00000000 00000013 00000000 6ff3be9f f9fde415
[<810462ec>] (imx_amp_power_init) from [<80101c7c>] (do_one_initcall+0x54/0x17c)
[<80101c7c>] (do_one_initcall) from [<81000ea0>] (kernel_init_freeable+0x16c/0x200)
[<81000ea0>] (kernel_init_freeable) from [<80ab0074>] (kernel_init+0x18/0x120)
```

```
[<80ab0074>] (kernel_init) from [<80108378>] (ret_from_fork+0x14/0x3c)
Code: 0a000005 e79ce103 e2833001 e794e10e (e5421004)
---[ end trace 0b1d1e3108025c69 ]---
Kernel panic - not syncing: Attempted to kill init! exitcode=0x0000000b
```

Running the examples

With all of that in place, we can now load the first example, a simple hello world on the Cortex-M4:

On the Cortex-A9:

```
# ./m4fwloader hello_world.bin 0x7f8000
```

Output from the Cortex-M4:

```
Hello World!
```

0x7f8000 is the address of the TCM. That is the memory from where the Cortex-M4 is running the code. The OCRAM or the regular DDR can also be used.

The first RPMSG example allows sending strings from the Cortex-A9 to the Cortex-M4 using a tty character device:

On the Cortex-A9:

```
# ./m4fwloader rpmsg_str_echo_freertos_example.bin 0x7f8000
virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-demo-channel addr 0x0
# insmod ./imx_rpmsg_tty.ko
imx_rpmsg_tty virtio0.rpmsg-openamp-demo-channel.-1.0: new channel: 0x400 -> 0x0!
Install rpmsg tty driver!
# echo test > /dev/ttyRPMSG
#
```

Output from the Cortex-M4:

```
RPMSG String Echo FreeRTOS RTOS API Demo...
RPMSG Init as Remote
Name service handshake is done, M4 has setup a rpmsg channel [0 ---> 1024]
Get Message From Master Side : "test" [len : 4]
Get New Line From Master Side
```

The other RPMSG example is a ping pong between the Cortex-A9 and the Cortex-M4:

On the Cortex-A9:

```
# ./m4fwloader rpmsg_pingpong_freertos_example.bin 0x7f8000
virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-demo-channel addr 0x0
# insmod ./imx_rpmsg_pingpong.ko
imx_rpmsg_pingpong virtio0.rpmsg-openamp-demo-channel.-1.0: new channel: 0x400 -> 0x0!
# get 1 (src: 0x0)
get 3 (src: 0x0)
get 5 (src: 0x0)
get 7 (src: 0x0)
get 9 (src: 0x0)
get 11 (src: 0x0)
get 13 (src: 0x0)
get 15 (src: 0x0)
get 17 (src: 0x0)
get 19 (src: 0x0)
get 21 (src: 0x0)
get 23 (src: 0x0)
get 25 (src: 0x0)
get 27 (src: 0x0)
get 29 (src: 0x0)
get 31 (src: 0x0)
get 33 (src: 0x0)
get 35 (src: 0x0)
get 37 (src: 0x0)
get 39 (src: 0x0)
```

Output from the Cortex-M4:

```
RPMSG PingPong FreeRTOS RTOS API Demo...
RPMSG Init as Remote
Name service handshake is done, M4 has setup a rpmsg channel [0 ---> 1024]
Get Data From Master Side : 0
Get Data From Master Side : 2
Get Data From Master Side : 4
Get Data From Master Side : 6
Get Data From Master Side : 8
Get Data From Master Side : 10
Get Data From Master Side : 12
Get Data From Master Side : 14
Get Data From Master Side : 16
Get Data From Master Side : 18
Get Data From Master Side : 20
Get Data From Master Side : 22
Get Data From Master Side : 24
Get Data From Master Side : 26
Get Data From Master Side : 28
Get Data From Master Side : 30
Get Data From Master Side : 32
Get Data From Master Side : 34
Get Data From Master Side : 36
Get Data From Master Side : 38
Get Data From Master Side : 40
```

Conclusion

While loading and reloading a Cortex-M4 firmware from Linux doesn't work out of the box, it is possible to make that work without too many modifications.

We usually prefer working with an upstream kernel. Upstream, there is a *remoteproc* driver, *drivers/remoteproc/imx_rproc.c* which is a much cleaner and generic way of loading a firmware on the Cortex-M4.

Author: Alexandre Belloni

Alexandre is a kernel and embedded Linux engineer at Bootlin, which he joined in 2013. His experience with embedded systems started much earlier! More details... View all posts by Alexandre Belloni

Alexandre Belloni / October 18, 2018 / Technical

3 thoughts on “Using the Cortex-M4 MCU on the i.MX6 SoloX from Linux”

olerem

October 26, 2018 at 8:09 pm

Hi,

here is a BSP which will build linux system for Cortex M4 and create firmware image from it. After this it will build linux system for Cortex A9 and include previously build firmware image.
https://git.pengutronix.de/cgit/ore/OSELAS.BSP-Pengutronix-DualKit/
Most/all of needed kernel parts are upstream. Communication between both systems is working over mailbox.

Raashid

August 21, 2019 at 12:40 pm

Hi,

Have you tried running the binary in U-boot using fatload? I don't see any console messages printed in the M4 debug console even after applying your board.c patch.

Raashid

August 22, 2019 at 7:41 am

Hi,

We can use the bootargs option "uart_from_osc" as mentioned in the FreeRTOS documentation to avoid patching board.c.
Thanks Alexandre Belloni for this article. Helped me a lot!!!

It would be great if you can any suggestions on how to run M4 firmware from U-Boot. The usual fatload procedure does not work. Does U-boot requires any patching?

Regards,
Raashid