FreeRTOS on FSiMX6SX Boards

Manual on how to use/configuring the software

Version 3.1
(2019-12-20)

About This Document

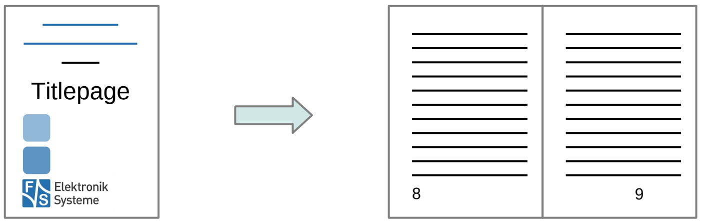
This document describes how to configure the Linux kernel, the device tree and the board to use it with FreeRTOS and its demo applications provided. The software is configured for the efusA9X, PicoCoreMX6SX and PicoCOMA9X from F&S under Linux.

Remark

The version number on the title page of this document is the version of the document. It is not related to the version number of any software release! The latest version of this document can always be found at http://www.fs-net.de.

How To Print This Document

This document is designed to be printed double-sided (front and back) on A4 paper. If you want to read it with a PDF reader program, you should use a two-page layout where the title page is an extra single page. The settings are correct if the page numbers are at the outside of the pages, even pages on the left and odd pages on the right side. If it is reversed, then the title page is handled wrongly and is part of the first double-page instead of a single page.



Typographical Conventions

We use different fonts and highlighting to emphasize the context of special terms:

`File names`

Menu entries

```
Board input/output
```

`Program code`

```
PC input/output
```

```
Listings
```

```
Generic input/output
```

`Variables`

History

Date	V	Platform	A,M,R	Chapter	Description	Au
2017-03-16	0.10	All	A	-	Title and “About this document” part created	TC
2017-03-22	0.20	All	A, M	1, 2, 3, 4	Modified title page (removed logos), Added Introduction, Added information on Installation, Added Configuration on U-Boot	TC
2017-03-23	0.30	All	A, M	4, 5, 7, 8	Added Modifying Device Tree chapter, Added FreeRTOS examples chapter, Added General Modifications chapter, Modified Appendix	TC
2017-03-23	0.40	All	A, M	6	Changed flexcan_network_epit subchapter, Added Description for gpio_imx	TC
2017-03-24	0.50	All	A, M	3, 5, 7	Explicit mentioning of armgcc download link, Changed download link for FreeRTOS, Added legal information, Claryfied path for examples	TC
2017-03-27	0.60	All	M	4, 7	Added information on enable_m4() to U-Boot modifying section, Changed information on ecspi examples	TC
2017-03-28	0.70	All	A, M	4, 7	Added information on changes in U-Boot regarding dram allocation, Changed information on rpmsg examples	TC
2017-03-30	0.80	All	A	5	Added information header regarding RPMsg	TC
2017-04-03	0.90	All	M	5, 7	Changed “modifications made” section for rpmsg examples, modified attention header to represent current state of development for RPMsg on boards with less than 1 GB RAM.	TC
2017-04-10	0.91	All	A, M	5, 7	Added subchapters to chapter 5.6 to clearify situation on RPMsg, changed “modifications made” part in RPMsg examples in chapter 8	TC
2017-04-12	0.95	All	A	7	Added more information in “changes needed” for the RPMsg pingpong examples	TC
2017-04-19	1.00	All	A, M	2, 3, 4, 7	Changed name for define in subchapter “Protecting modules”, Added new description for i2c examples from NXP, Added own examples, changed some naming conventions for paths, Added attention header for GPIOs	TC
2017-04-21	1.10	All	M	7	Changed command for booting m4, Changed attention header in chapter 3 regarding the current state of development	TC
2017-04-25	1.20	All	A, M	-	Modified title and “About this Document” to reflect changes on PicoCOMA9X	TC
2017-05-05	1.30	All	A, M	2, 3, 4, 5, 6, 7	Changed structure of pin assignments chapter; moved the tables from chapter 6 to 2 and modified their content, changed refferences to tables in chapter 6, mentioned picocoma9x.dts additionally, Added “Known Issues” section to appendix, added information about interfering audio chip and RPMsg	TC
2017-05-08	1.35	All	M	4, 5	Changed “Changes in gpio_pins.[ch]” to reflect newes development regarding the configure_gpio_pin() function	TC
2017-05-10	1.40	All	A, M	6	Added subchapter about resetting Cortex-M4, modified information about i2c demo	TC
2017-05-30	1.41	All	M	4	Fixed typo in path to file	TC
2017-05-31	1.50	All	A, M	5, 6, 7	Changed information on RPMsg VRING allocation addresses to reflect recent changes in the build system, added chapter about new build process, Moved FreeRTOS examples from chapter 6 to 7	TC
2017-06-07	1.60	All	A, M	2, 6	Added title to pin tables to clarify the board name, modified information regarding build/make/install and clean	TC
2017-06-20	1.61	All	M	2	Fixed wrong interface for efusa9x GPIOs	TC
2017-06-23	1.65	All	A, M	1, 7	Added figures and an informational text about time measurement examples provided by F&S	TC
2017-06-29	1.66	All	A	1	Added figure for performance scaling govenor, added result text at the end of chapter 1	TC
2017-07-03	1.67	All	M	1	Modified screenshots to a more unified measuring scale	TC

2017-07-05	1.68	All	M	1	Swapped images; Trigger set on Input; Input signal is upper one while output is on the lower part of the images	TC
2017-07-06	1.69	All	M	1	Fixed minor spelling mistakes	TC
2017-07-10	1.70	All	A, M	1	Removed powersaving figures, added another set of figures	TC
2017-07-11	1.71	All	M	1	Modified Screenshots: set legend to the left corner, resized images	TC
2017-07-17	1.72	All	M	5, 6	Added information on changes made to board.h, added more details in building with prepare.sh	TC
2017-07-21	1.73	All	M	4	Modified chapter 4.1 to reflect current state of development, added new subchapter 4.2 for bootaux usage	TC
2017-07-27	1.74	All	M	6	Merged 6.3 with 6.2	TC
2017-08-03	1.75	All	A	7	Added chapter on custom board creation	TC
2017-08-04	1.80	All	A	2, 4, 5, 8	Added subsection about ADC Pins on I2C-Ext. added subsection regarding bootaux, modified subsection about board.h regarding BOARD_I2C_MODULE_ADDR, added hint about building examples, changed the names for mutliple demos (F&S)	TC
2017-08-10	1.81	All	M	2	Fixed wrong gpio for efusa9x	TC
2017-08-21	1.82	All	A, M	8	Inserted attention header for RPMsg enabling in device tree, shortened ocram address, removed unimplemented demo	TC
2017-08-22	1.83	All	M	8	Changed some parts inside chapter 8, removed outdated subchapter about resetting Cortex-M4	TC
2017-08-23	1.84	All	M	2, 8	Moved remark about unported examples to chapter 8, gpioSpiACs1 is the default LED for efusa9x now, made those pesky references working again	TC
2017-08-25	1.85	All	A, R	8	Added subchapter about fs_adc_i2c_rpmsg_demo, removed entry in ADC table for external reference voltage	TC
2017-08-30	1.86	All	M,R	4, 5, 6, 8	Changed subchapter about enabling RPMsg node, removed subchapter VRING allocation addresses, added/changed information about protecting modules , fixed attentipn header regarding location of BSP	TC
2017-08-31	1.87	All	M	3	Changed path for FreeRTOS BSP	TC
2019-12-20	1.88	All	M	4,6,8	Fixed minor spelling mistakes. Reworked fs_adc_i2c_rpmsg_demo	PG
2017-09-19	1.88	All	M	8	Fixed defines in blinking_imx example	PG
2017-09-22	1.88	All	M	8	Added rpmsg driver workaround	PG
2017-09-26	1.88	All	M	4	Changed description of SUPPOR_M4 define	PG
2017-10-04	1.88	All	M	2.4	Changed I2C Pins to I2C3/I2CB	PG
2017-10-10	1.88	All	A	1	Added RPMsg measurements	PG
2017-10-11	1.88	All	M	2	Reworked GIO Pin assignment	PG
2017-10-17	1.88	All	A	1	Added FreeRTOS form diff. Memory regions	PG
2017-10-17	1.88	All	M	8.2.2	Changed example back to normal OCRAM	PG
2017-10-17	1.88	All	A	1	Added FreeRTOS form diff. Memory regions	PG
2018-01-10	2.0	All			Release version 2.0	PG
2018-03-27	2.1	All	A,M	8.2.5	Add hello_world_split example, improve misspelling	PJ
2019-01-23	3.0	All	M	All	Reworked freertos-SDK, splitted documentation into 2 separate files. Added pin_mux-tables from extern excel file. Added support for picocoremx6sx	PG
2019-12-20	3.1	All	M	M	Restructure of the document	PJ

V	Version
A,M,R	Added, Modified, Removed
Au	Author

Table *of Contents*

1 Pin Assignment

In the following subchapters you can find an overview which pins are used for each Board. The examples itself also contains the necessary pins.

1.1 efusA9X

1.1.1 GPIO

Set as	Function	Device	GPIO	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
KEY	I2C_B_IRQ	I2C3_IRQn	GPIO7_IO01	J4_86	J22_48
LED, BLINK	PWM_A	PWM5	GPIO3_IO24	J4_25	J22_32

1.1.2 FLEXCAN

Function	Device	efusA9X Rev 1.2X	Function	efus-SINTF Rev 1.30
CAN_B_TX	FLEXCAN2	J4_35	CAN_B_H	J22_55
CAN_B_RX	FLEXCAN2	J4_37	CAN_B_L	J22_56

1.1.3 ECSPI

Function	Device	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
SPI_B_MISO	SPI1	J4_50	J22_23
SPI_B_MOSI	SPI1	J4_52	J22_24
SPI_B_CLK	SPI1	J4_54	J22_25
SPI_B_CS1	SPI1	J4_56	J22_26

1.1.4 I2C

Function	Device	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
I2C_B_SCL	I2C3	J4_84	J22_46
I2C_B_SDA	I2C3	J4_82	J22_45

1.2 PicoCOMA9X

1.2.1 GPIOs

Set as	Function	Device	GPIO	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
KEY	SPI_MISO	SPI1	GPIO2_IO11	J2_26	J10_3	J3_3
LED, BLINK	SPI_MOSI	SPI1	GPIO2_IO15	J2_29	J10_6	J3_6

1.2.2 FLEXCAN

Function	Device	PicoCOMA9X Rev 1.10	Function	PC2-SINTF	PCOMnet
CAN_A_TX	FLEXCAN1	J2_30	CANH	J10_25	J3_13
CAN_A_RX	FLEXCAN1	J2_31	CANL	J10_26	J3_14

1.2.3 ECSPI

Function	Device	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
SPI_MISO	SPI1	J2_26	J10_3	J3_3
SPI_MOSI	SPI1	J2_29	J10_6	J3_6
SPI_CLK	SPI1	J2_28	J10_5	J3_5
SPI_CS0	SPI1	J2_27	J10_4	J3_4

1.2.4 I2C

Function	Device	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
I2C_A_SCL	I2C4	J2_33	J10_10	J3_10
I2C_A_SDA	I2C4	J2_32	J10_9	J3_9

1.3 PicoCoreMX6SX

1.3.1 GPIOs

Set as	Function	Device	GPIO	PicoCoreMX6SX Rev 1.10	PCoreBBRGB Rev 1.20
KEY	GPIO4_25	GPIO	GPIO4_IO25	J1_26	J10_4
LED, BLINK	GPIO4_26	GPIO	GPIO4_IO26	J1_24	J10_3

1.3.2 FLEXCAN

Function	Device	PicoCoreMX6SX Rev 1.10	Function	PCoreBBRGB Rec 1.20
CAN_A_TX	FLEXCAN1	J1_12	CAN_1_H	J6_4
CAN_A_RX	FLEXCAN1	J1_10	CAN_1_L	J6_3

1.3.3 ECSPI

Function	Device	PicoCoreMX6SX Rev 1.20	PCoreBBRGB Rec 1.20
SPI_A_MISO	ECSPI5	J1_16	J10_17
SPI_A_MOSI	ECSPI5	J1_18	J10_16
SPI_A_CLK	ECSPI5	J1_20	J10_12
SPI_A_SS0	ECSPI5	J1_14	J10_14

1.3.4 I2C

Function	Device	PicoCoreMX6SX Rev 1.20	PCoreBBRGB Rec 1.20
I2C_A_SCL	I2C4	J1_21	n.c.
I2C_A_SDA	I2C4	J1_23	n.c.

1.4 I2C-Extension-Board

1.4.1 I2C

The pin assignment for I2C requires the jumper JP1 / JP2 / JP3 / JP5 are set (NetDCU/PicoMOD).

Name	Connector
I2C_SCL	J1_11
I2C_SDA	J1_10

1.4.2 ADC

Name	Connector
CH0	J2_17
COM	J2_25

2 Different kind of memory regions

The Cortex-M4 code can be executed from different memory regions, as demonstrated in the “`hello_world_`” examples. On the efusA9X, PicoCoreMX6SX and the PicoCOMA9X it supports 3 different kind of memory chips: TMC, OCRAM and DRAM (QSPI is not supported on these boards).

In most cases the on-chip TCM (Tightly-Coupled-Memory) is used, which is the fastest and safest way to run Cortex-M4 applications, because only the Cortex-M4 has access to it and therefor it needs no further protection. Also using the TCM may help meeting possible real-time requirements, because an access to a different memory chip may be delayed if the bus is currently used by the Cortex-A9.

But TCM is restricted to 32 KB Instructions and 32 KB data. If your application is too big you might have to move it to another memory chip, which requires further considerations.

The Cortex-M4 implements a modified Harvard memory architecture, meaning that instruction and data fetches are made over separated bus port if accessed with different address ranges. Instructions should only be placed in the address range from 0x00000000 to 0x1FFFFFFF whereas data should be placed from 0x20000000 to 0xFFFFFFFF. This allows parallel fetching instruction and data.

The TCM is separated into a 32 KB TCML for instruction and a 32 KB TCMH for data, which reflects the Harvard architecture of the Cortex-M4. If you want to place you code in a different memory region you will have to consider this separation by using the right addresses for instructions and data (see table below).

It is possible to place instructions into the data address range. But then the Cortex-M4 needs twice as much cycles to fetch an instruction.

Memory	A9 Address	M4 Address	Size (Type)
DRAM	0x80000000	0x80000000	1.5 GB (Data)
DRAM aliased	0x80000000	0x10000000	256 MB (Instruction)
OCRAM	0x00900000	0x20900000	128 KB (64 KB) (Data)
OCRAM aliased	0x00900000	0x00900000	128 KB (64 KB) (Instructions)
TCMH	0x00800000	0x20000000	32 KB (Data)
TCML	0x007F8000	0x1FFF8000	32 KB (Instructions)

Table 1: A9, M4 Memory map

Also you will have to protect the memory regions used by the Cortex-M4 against access form the Cortex-A9. On the M4 side this is done by the RDC (Remote Domain Controller), which will block any access from Cortex-A9, resulting in a Linux kernel panic. To prevent this the memory regions used by the Cortex-M4 need to be excluded from the memory map of the Cortex-A9.

This can be done by setting the U-Boot environment variable:

```
#setenv reserved_ram_size 0x100000
```

This will reserve the last 1MB[1] of the first 256MB of the DRAM, which corresponds to the aliased DRAM for instructions of the Cortex-M4. The addresses to access this area are:

- 0x1ff0_0000 to 0x2000_0000 for instructions
- 0x8ff0_0000 to 0x9000_0000 for data

You can change the position of the reserved memory area by setting the the U-Boot environment variables

```
#setenv reserved_ram_base 0x…

#setenv reserved_ram_end 0x…
```

- Setting base and size will reserve you $base to $base+$size
- Setting end and size will reserve you $end-$size to $end

1 Right now the U-Boot always rounds up to the next MB. So there will be 1MB reserved by default

Different kind of memory regions

- Setting base and end will reserve you $base to $end. The size-variable will be ignored.
- Setting just base ore just end is invalid. Nothing will be reserved.

The first 64 MB of the DRAM are reserved for the Linux kernel. If you try to place your reserved memory area there or behind the maximum available ram, it will be cropped.

Also keep in mind that placing the Corex-M4 instruction code behind the first 256 MB, will result in slower code execution and may cause other problems.

If you do not want to reserve any memory at all you can set all the `reserved_ram_*` environment variables to 0.

Attention

If you are starting an M4-DRAM example from the U-Boot, make sure that you don't override it by loading anything big into the DRAM, afterwards.

You can set the $loadaddr environment variable behind the M4-code area to load big files.

RPMsg

By default the U-Boot will reserve 64 KB at the end of your reserved memory area for the RPMsg vring buffers. If you change the location of you reserved memory area you will have to run the prepare.sh script again and set the new end address of your reserved memory area.

You will have to make sure, that your M4 program does not use these last 64KB if it runs from the DRAM.

If you do not reserve any DRAM memory you will have to disable RPMsg in you device tree.

Also you will have to activate the comment at the beginning of your *efusa9x.dts*, *picocoma9x.dts* or *picocoremx6sx.dts* file. It will apply the following changes:

OCRAM

Adds:

```
&ocram {
            reg = <0x00901000 0xf000>;
}
```

This will split the OCRAM so that only the lower 60 KB will be used by Linux.

4KB will be needed for the low power examples and are reserved in the *&clks* node:

```
fsl,shared-mem-addr = <0x91F000>;
fsl,shared-mem-size = <0x1000>;
```

The rest can be used by the Cortex_M4

DRAM
The last 64 KB of your DRAM will be used for RPMsg:

```
&rpmsg {
    vdev-nums = <1>;
    /* This will be overitten by the U-Boot */
    reg = <0xBFFF0000 0x10000>;
    status = "okay";
};
```

Changes needed in the FreeRTOS package

If you want to run a Cortex-M4 application from a different memory type, go to your applications `CmakeLists.txt` file:

```
examples/fsimx6sx/demo_apps/YOUR_APPLICATION/armgcc/CMakeLists.txt
```

And change the linker file to your required memory type:

```
MCIMX6X_M4_{tcm/ocram/ddr}.ld
```

If you want to change the actual addresses, go to the linker file:

```
platform/devices/MCIMX6X/linker/gcc/MCIMX6X_M4_XXX.ld
```

and change these hard-coded addresses:

```
/* Specify the memory areas */
MEMORY
{
   m_interrupts   (RX)  : ORIGIN = 0x1ff00000, LENGTH = 0x00000240
   m_text         (RX)  : ORIGIN = 0x1ff00240, LENGTH = 0x00007DC0
   m_data         (RW)  : ORIGIN = 0x8ff08000, LENGTH = 0x00008000
}
```

Make sure to use the right addresses for instruction and data and to protect your memory from Linux accesses.

Finally load the code to your starting address ad run it.

3 Installation

This section describes the installation of the CST code-signing client files.

3.1 Installation of the GCC embedded toolchain

The examples are tested and can be built with the GCC embedded toolchain (gcc-arm-none-eabi-8-2019-q3-update), which can be found under https://launchpad.net/gcc-arm-embedded.

If the toolchain is not installed, you have to download the file and extract the content to your filesystem:

```
tar -xvjf gcc-arm-none-eabi-${version}.tar.bz2
```

where ${version} will be replaced by the corresponding version you've downloaded.

It is necessary to export the `ARMGCC_DIR` environment variable, if it´s not already exported:

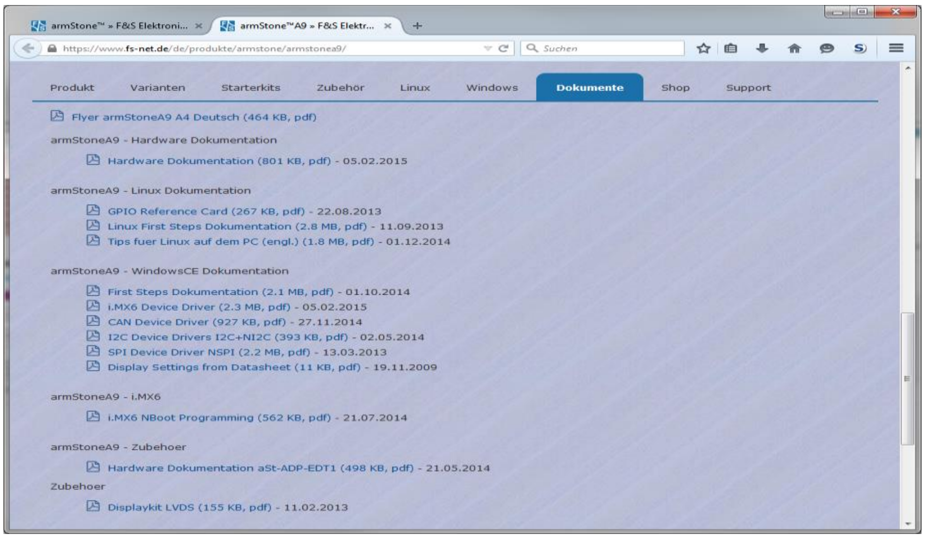
```
export ARMGCC_DIR=/usr/local/arm/gcc-arm-none-eabi-${version}
```

For a more convenient way you can add this to the rc file of your favorite shell (e.g. zshrc, bashrc, etc.)

3.2 Download Source Code

To download FreeRTOS source code, go to the F&S main website

http://www.fs-net.de


armStone™ » F&S Elektroni...
armStone™A9 » F&S Elektr...
https://www.fs-net.de/de/produkte/armstone/armstonea9/
Suchen
Produkt
Varianten
Starterkits
Zubehör
Linux
Windows
Dokumente
Shop
Support

Flyer armStoneA9 A4 Deutsch (464 KB, pdf)

armStoneA9 - Hardware Dokumentation

Hardware Dokumentation (801 KB, pdf) - 05.02.2015

armStoneA9 - Linux Dokumentation

GPIO Reference Card (267 KB, pdf) - 22.08.2013
Linux First Steps Dokumentation (2.8 MB, pdf) - 11.09.2013
Tips fuer Linux auf dem PC (engl.) (1.8 MB, pdf) - 01.12.2014

armStoneA9 - WindowsCE Dokumentation

First Steps Dokumentation (2.1 MB, pdf) - 01.10.2014
i.MX6 Device Driver (2.3 MB, pdf) - 05.02.2015
CAN Device Driver (927 KB, pdf) - 27.11.2014
I2C Device Drivers I2C+NI2C (393 KB, pdf) - 02.05.2014
SPI Device Driver NSPI (2.2 MB, pdf) - 13.03.2013
Display Settings from Datasheet (11 KB, pdf) - 19.11.2009

armStoneA9 - i.MX6

i.MX6 NBoot Programming (562 KB, pdf) - 21.07.2014

armStoneA9 - Zubehoer

Hardware Dokumentation aSt-ADP-EDT1 (498 KB, pdf) - 21.05.2014

Zubehoer

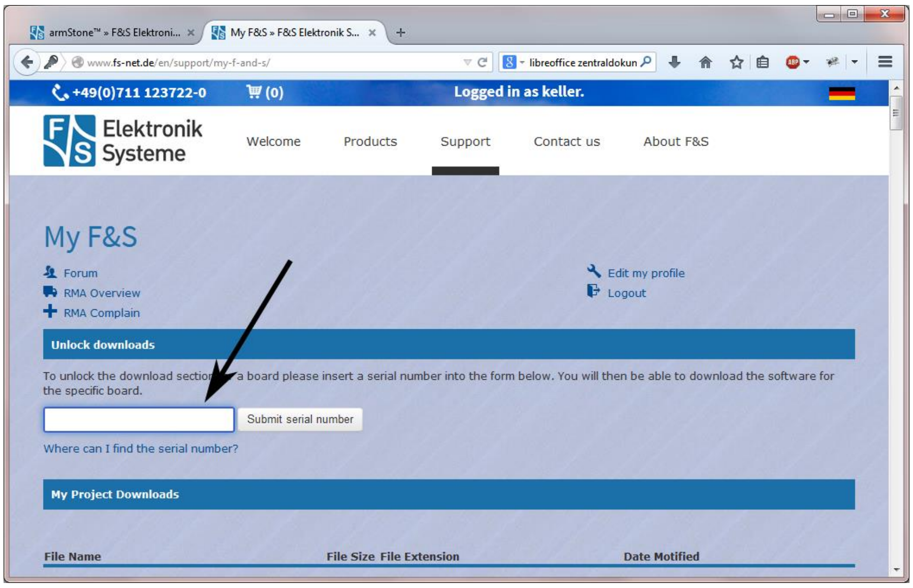
Displaykit LVDS (155 KB, pdf) - 11.02.2013

Figure 1: Register with F&S website

First you have to register with the website. Click on *Login* right at the top of the window and on the text “I am not registered, yet. Register now” (*Figure 1*).

In the screen appearing now, fill in all fields and then click on *Register*. You are now registered and can use the personal features of the website, for example the Support Forum and downloading software.

After logging in, you are at your personal page, called “My F&S”. You can always reach this place by selecting *Support → My F&S* from the top menu. Here you can find all software downloads that are available for you. In the top sections there are private downloads for you or your company (may be empty) and in the bottom section you will find generic downloads for all registered customers.



To unlock the download section for a board please insert a serial number into the form below. You will then be able to download the software for the specific board.

Figure 2: Unlock software with the serial number

To get access to the software of a specific board, you have to enter the serial number of one of these boards (see *Figure 2*). Click on “Where can I find the serial number” to get pictures of examples where to find this number on your product. Enter the number in the white field and press *Submit serial number*. This enables the software section for this board type for you. You will find Linux, Windows CE, and all other software and tools available for this platform like `DCUTerm` or `NetDCUUsbLoader`.

First click on the type of your board, e.g. efusA9X, then on Linux. Now click on FreeRTOS. This will bring up a list of all our FreeRTOS releases. Old releases up to 2019 had V<x>.<y> as version identifier, new releases use V<year>.<month>. We will abbreviate this as <v> from now on. Select the newest version, for example *freertos-fsimx6sx-V2019.12*. This will finally show two archives that can be downloaded.

When you look at our Linux releases, you will find a list of all our releases and a README text. There are usually two files related to a release.

`freertos-fsimx6sx-<v>.tar.bz2`	This is the main release itself containing all sources, the binary images, the documentation and the toolchain.

3.3 Release Content

These tar archives are compressed with bzip2. To see the files, you first have to unpack the archives

```
tar xvf freertos-<arch>-<v>.tar.bz2
```

This will create a directory `<arch>-<v>` that contains all the files of the release. They often use a common naming scheme:

`<package>-<platform>-<v>.<extension>`

With the following meaning:

`<package>`	The name of the package (e.g. `freertos-bsp`). If it is a source package, we also add the version number of the original package that our release is based on, for example `freertos-bsp-1.0.1`.
`<platform>`	The name of a board, if the package is only valid on one board (e.g. `efusA9X`); or the name of an architecture, if the package is valid on different boards of the same architecture (e.g. `fsimx6sx`), or the string `f+s` or `fus` if the package is architecture independent.
`<v>`	Release version, consisting of a letter `V` for version and the year and month of the release (e.g. `V2019.12`).
`<extension>`	The extension of the package (e.g. `.bin`, `.tar.bz2`, etc.).

The following table lists the files that you get after unpacking the release archive. To avoid having a too excessive list, we use the wildcard `*` in some entries to refer to a whole group of similar file names that only differ in the name of the board or module.

Directory/File	Description
`/`	**Top directory**
`README-freertos-f+s.txt`	Release information (FreeRTOS)
`setup-freertos`	Script to unpack FreeRTOS source packages to a build directory
`binaries/`	**Images to be used with the board directly**
`efusa9x/*-<v>.bin`	Precompiled examples for efusa9x
`picocoma9x/*-<v>.bin`	Precompiled examples for picocoma9x
`picocoremx6sx/*-<v>.bin`	Precompiled examples for picocoremx6sx
`sources/`	**Source packages**
`freertos-bsp-1.0.1-fsimx6sx-V2019.12.tar.bz2`	FreeRTOS source
`toolchain/`	**Cross-compilation toolchain**
`gcc-arm-none-eabi-8-2019-q3-update-linux.tar.bz2`	ARM toolchain to use with `<arch>`
`doc/`	**Documentation**
`FreeRTOS_on_FSiMX6SX_Boards_eng.pdf`	Manual on how to use/configuring the software

Table 2: Content of the created release directory

3.4 Unpacking the Source Code

The source code packages are located in the `sources` subdirectory of the release archive. We will assume that you want to create a separate build directory where you extract the source code and build all the software.

We have prepared a shell script called `setup-freertos` that does this installation automatically. Just call it when you are in the top directory of the release and give the name of the build directory as argument.

```
cd <release-dir>
./setup-freertos <build-dir>
```

Add option `--dry-run` if you want to check first what this command will do. Then only a list of actions will be output but no actual changes will take place. For further information simply call

```
./setup-freertos --help
```

If you prefer to do the installation by hand, well, the script more or less executes the following commands, just with some more checks and directory switching.

```
mkdir <build-dir>
tar xf freertos-bsp-1.0.1-fsimx6sx-<v>.tar.bz2
```

4 Description of the FreeRTOS examples directory structure

The examples directory contains the SoC and board specific Cortex-M4 examples. The first level distinguishes between the different SoC-architectures. At the second level you will find the SoC specific examples. For the MX6SX-examples the board specific examples are located in the *board_specific_files* directory.

The examples are structured as follows:

4.1.1 demo_apps

Here you can find the applications which highlight certain key features of the ARM Cortex-M4 Core combined with FreeRTOS.

4.1.2 driver_examples

You can find simple applications here which are intended to show peripheral drivers working with FreeRTOS in the bare metal environment.

4.1.3 multicore_examples

Here you can find examples, which demonstrate the multicore communication via RPMsg`.`

4.1.4 board_specific_files

This folder contains board specific files like pinout. When you execute the prepare.sh script, links to the specific board files will be created in the freertos-`fsimx6sx-<v>/boards/fsimx6sx/ directory.`

Attention

Examples which make use of on-board sensors, e.g. magnetic field strength measuring were not ported, so use them at your own risk!

Note 1: Information about not ported examples

5 Configuration for Cortex-M4 usage

5.1 Changes regarding official U-Boot

F&S provides you with a modified U-Boot which can make use of the Cortex-M4 via the `bootaux` command. Since our U-Boot is heavily modified compared to the official release from NXP, it's not advisable to use any other than the one provided by F&S.

1. We modified the `bootaux` command. You can enable, disable and start the auxiliary core by using the commands described in the next subchapter.

If you want to boot your examples from the dram or try RPMsg examples, make sure the `reserved_ram_size` variable is set.

```
setenv reserved_ram_size 100000
```

5.2 Using bootaux

Active or not?

If you want to know if the auxiliary core is running, just type

```
bootaux
```

This will state if the clocks are active and the core available.

Simple start

Using the auxiliary core can be achieved by using the following command line inside of the U-Boot environment:

```
tftp ${m4_file}; cp.b $loadaddr 0x7f8000 $filesize; bootaux
0x7f8000
```

This will load an image defined by `m4_file` via tftp to your board, move it to the TCM and start the auxiliary core.

Reloading of auxiliary core image

If you need to reload or switch the image or if you simply want to deactivate the auxiliary core, just type

```
bootaux stop
```

To re-enable the auxiliary core type

```
bootaux start
```

Then you can issue a

```
tftp ${m4_file}; cp.b $loadaddr 0x7f8000 $filesize; bootaux
0x7f8000
```

To simply reload an image from your tftp server and execute it on the auxiliary core (Make sure to modify `m4_file` according to the name of the image you want to execute).

5.3 Modifying the Linux Device Tree

To get all the examples running the Linux Device Tree has to be changed. By uncommenting

```
#define SUPPORT_M4
```

In the device tree all the necessary changes will be made.

6 General Modifications on FreeRTOS examples

The files needed for general modifications (`board.*`, `pin_mux.*`, etc.) can be found under `freertos-fsimx6sx-<v>/examples/fsimx6sx/board_specific_files/BOARD_NAME`.

6.1 Changes to the clock handling

In order to boot Linux while running the FreeRTOS examples, the Cortex-M4 has to tell the Cortex-A9, which clocks are needed and shouldn't get turned off. This is done by a shared memory area in the OCRAM which needs to get initialized from both sides. This has not been done at the FreeRTOS side so far.

We implemented a functionality that initializes the shared memory in the *hardware_init()* function of each example and adds an entry to the shared clock table whenever a shared clock gets enabled or disabled .

6.2 Changes in board.c

We removed unnecessary parts in *BOARD_ClockInit()* and *dbg_uart_init()* since some of these settings were already covered by our NBoot.

We added a function that initializes the shared clock memory and gets called by the *BOARD_ClockInit()* function.

Also the *dbg_uart_init()* function now enables the UART shared clock nodes.

6.3 Changes in board.h

- We modified the `BOARD_GPIO_KEY_*` and `BOARD_GPIO_LED_*` defines to comply with our GPIOs on the board.
- The `BOARD_ADC_INPUT_CHANNEL` was changed to comply with our ADCs on the board.
- `BOARD_I2C_MODULE_ADDR` was added for i2c-demos
- The pin mux configuration was a bit tedious, so we added a macro which can be used in conjunction with

  `./devices/MCIMX6X/include/imx6sx_iomuxc_pins.h`

  which resembles a file from the Linux kernel to set the pad settings. This simplifies the setting of the pads and mux's in `pin_mux.c` and coherent with the Linux ones used by the Cortex-A9
- `BOARD_I2C_MODULE_ADDR` was added. This is the slave address for the I2C*X* module used on the board

- added defines for sema4 gate number and shared memory ocram start address
- added defines for board specific shared clock indexes

Macro for setting the pad

`set_iomux(PAD_MUX_SETTINGS, pad_value)`

`PAD_MUX_SETTINGS ...........`	`Array of 5 values which corresponds to the iomuxc values stated in imx6sx_iomuxc_pins.h`
`pad_value ..................`	`hex value representing the settings for the pad register`

Example

```
/* Setting the TXD for UART1 */
set_iomux(MX6SX_PAD_GPIO1_IO04__UART1_TX, 0x1B0B1);
```

This will set the `UART1_TX` to 0x1B0B1 in the IOMUXC.

6.4 Changes in gpio_pins.c

We added predefined GPIOs for the `blinking_imx_*` and other demos. The unused ones were commented out since we modified the ported examples.

The *configure_gpio_pin()* was changed. This decision was made since this code seems a bit confusing; this part should be done in the *pin_mux.c* because it's similar to setting the pads and mux's in the IOMUXC. We use the *set_iomux()* macro in this function instead.

6.5 Changes in gpio_pins.h

We modified the *_gpio_config* struct to reflect the changes made for the *configure_gpio_pin()* function.

Some of the *extern gpio_config_t* declarations were commented out since we changed the predefined GPIOs.

6.6 Changes in pin_mux.c

Instead of setting the Pads, Mux's and everything else via the NXP defines we make use of the *set_iomux()* macro described in chapter 6.3 to simplify the whole setup process.

We mainly changed the pads and mux's to the ones used on the efusA9X board from F&S.

6.7 Changes in ccm_imx6sx.h

Added function that makes the necessary shared clock entries when a clock gets enabled.

7 Compiling the examples

To simplify the process of building, configuring the examples and cleaning up we provide you with a set of bash scripts located in the root directory of the FreeRTOS BSP:

7.1 Prepare.sh

This script will configure board relevant settings and create symlinks to the board specific header files in `examples/fsimx6sx.` You can execute the script in your terminal by typing

```
./prepare.sh
```

and follow the instructions given by the cli:

```
Choose on of the following boards for which you want to build the examples:
efusa9x[1]   picocoma9x[2]  picocoreMX6SX[3]  picocore7ulp[4] …[…]
Enter number in []-brackets for the corresponding board: 2
Symlinks to board specific files created!
Do you want a Release or Debug build?
 (r/d) [default: r]: r
Did you change the address of the reserved DRAM-memory?
 (y/n) [default: n]: n
Setting vring_buffers to 0x8fff0000
If you have a PicoCOMA9X with a fused PCIE_DISABLE, you need
the workaround to disable RDC_* calls, otherwise your CPU will hang
(See documentation, last chapter)
(y/n) [default: n]: y
All set up, starting cmake...
```

Note

The question in prepare.sh “Did you change the address of the reserved DRAM-memory?” was added. We changed the memory area where the ring buffers are located. For more information please take a look at chapter 2 Different kind of memory regions.

7.2 Make

The `prepare.sh` script will configure and invoke `cmake` to generate a Makefile. After this, you can run

```
make -jN
```

To build all examples located in `boards/fsimx6sx` and install the binaries to `freertos-bsp-1.0.1-fsimx6sx-V<year>.<month>/bin/$BOARD.`

`N` is the number of the CPU cores in your development PC.

If you want to build a specific example just type

```
make -jN example_name && make install/fast
```

to build and install the binary of the chosen example.

Type

```
make help
```

for a list of possible examples for make.

By executing

```
make clean-all
```

you can clean up all build files and binaries. This will be necessary if you make changes to the `CmakeLists.txt` in the root directory of the FreeRTOS BSP.

8 Adding custom boards

If you're using a custom board, you have to tell the `prepare.sh` script about its existence and create some configuration files (or simply copy the existing ones).

To tell the script about it, change the following lines in the `prepare.sh` script:

```
declare -a SUPPORTED_BOARDS=("efusa9x" "picocoma9x" "pico-
coremx6sx" “picocoremx7ulp” "boardname")
```

where *boardname* represent the name of your board and an entry to

```
declare -a SUPPORTED_SOCS=("fsimx6sx" "fsimx6sx" "fsimx6sx"
"fsimx7ulp" fsimx6sx)
```

so the number of your *boardname* matches the number of its specific SOC.

The following files, located at `examples/fsimx6sx/board_specific_files/`*`boardname`*, are needed to successfully compile the BSP for your own board:

- *`boardname`*`_board.c`
- *`boardname`*`_board.h`
- *`boardname`*`_pin_mux.c`
- *`boardname`*`_pin_mux.h`
- *`boardname`*`_gpio_pins.c`
- *`boardname`*`_gpio_pins.h`

boardname must be replaced by the name of your board. This must be same name as uses in the *SUPPORTED_BOARDS* array used in the `prepare.sh` script.

9 FreeRTOS examples

In this chapter we will provide you with necessary information on the demo and driver applications.

The "**Description**" will inform you about the demo's purpose.

In the "**Modifications made**" section you will find useful information if changes were made to certain files by F&S and the reason behind these changes.

"**Changes needed**" is the most important section. You will find the information necessary to successfully build and execute the examples here.

The last section, "**Execute binary**" will tell you the required steps to execute the image built.

9.1 General build and run information

Build the examples like described in **Building the examples** and copy them to you tftp-directory.

It might be helpful to define a shorthand command to run the Cortex-M4 examples inside the U-Boot:

```
setenv m4 "'tftp ${m4_file}; cp.b $loadaddr 0x7f8000 $filesize;
bootaux 0x7f8000'"
```

In the following examples we will use this variable `m4` instead of the content of this variable, except the destination address is not equal to the TCM address.

Make sure that the environment variable `reserved_ram_size` in the U-Boot is set to 1MB

```
setenv reserved_ram_size 100000
```

Now simply run the commands described in the **Execute binary** section of each example.

Attention

If you want to run the examples while booting the Cortex-A9, you have to make sure that the define `SUPPORT_M4` in your Linux device-tree is uncommented.

To use the CortexM4 in the U-Boot and Linux, without complications, we need to change the clock frequency from 80Hz to 24Hz. To achieve that, we can simply add an argument in our U-Boot: "setenv extra uart_from_osc".

9.2 demo_apps

Remark

The documentation is based on the FreeRTOS BSP 1.0.1 package from NXP.

Some of the software examples provided by NXP expect a certain module or sensor to be available on the board. Since F&S boards do **NOT** provide these, the associated examples weren't ported at all.

9.2.1 hello_world

Description

Just a simple application which prints a “hello world!” string on the Cortex-M4 side and echoing input back to the user if he is connected to the Cortex-M4 via UART.

Modifications made

None.

Changes needed

None.

Execute binary

Run

```
setenv m4_file "hello_world.bin"
run m4
```

to start the example.

9.2.2 hello_world_ocram

Description

This example doesn't use the TCM but can be launched in OCRAM instead.

Modifications made

None

Changes needed

None

Execute binary

```
setenv m4_file "hello_world_ocram.bin"
tftp hello_world_ocram.bin; cp.b $loadaddr 0x910000 $filesize;
dcache flush; bootaux 0x910000
```

9.2.3 hello_world_ddr

Description

Works similar to `hello_world_ocram` except that it uses DDR instead of OCRAM.

Modifications made

None

Changes needed

Make sure that the environment variable `reserved_ram_size` in the U-Boot is set to 1MB

```
setenv reserved_ram_size 100000
```

Execute binary

Run

```
setenv m4_file "hello_world_ddr.bin"
tftp hello_world_ddr.bin; cp.b $loadaddr 0x8ff00000 $filesize;
dcache flush; bootaux 0x8ff00000
```

9.2.4 hello_world_qspi

This example wasn't ported because we do not offer a QSPI chip on the efusA9X and Pico-COMA9X.

If you want to run this example you have to port it by yourself.

9.2.5 hello_world_split

Description

Just a simple application which prints the name of the current function on the Cortex-M4 side and echoing input back to the user if he is connected to the Cortex-M4 via UART. If the user types in small letter “s” then program is switching between the functions. The special on this example is that, it have 3 different functions (*HelloTaskTCM*, *HelloTaskDRAM*, *Hello-TaskOCRAM*). These functions are located in 3 different memory areas. So this example shows you how to split your code and outsource different functions to different memory areas. Normally the text section of the DRAM should be located in the first 256 MB. This makes the Code a little bit faster, but the U-Boot doesn’t support this at the moment. So that’s why we use also the upper address area, like the data section of the DRAM.

Modifications made

The `hello_world` example was the basis of this example. Rename main function and improve it. Add 2 more C-Files for DRAM and OCRAM. Created linker Script called `MCIMX6X_M4_split.ld` and created Startup file called `startup_MCIMX6X_M4_split.S`. Modified `CMakeLists.txt`.

Changes needed

Make sure that the environment variable `reserved_ram_size` in the U-Boot is set to 1MB

```
setenv reserved_ram_size 100000
```

Image

You have 3 different binaries as output files. `hello_world_split_tcm.bin`, `hello_world_split_dram.bin` and `hello_world_split_ocram.bin`.

Execute binary

Run

```
tftp hello_world_split_tcm.bin
cp.b $loadaddr 0x7f8000 $filesize
tftp hello_world_split_dram.bin
cp.b $loadaddr 0x8ff00000 $filesize
tftp hello_world_split_ocram.bin
cp.b $loadaddr 0x910000 $filesize
bootaux 0x7f8000
```

to start the example.

9.2.6 blinking_imx_demo

Description

By running this demo you can let a LED blink or print out '+' and '-' with different frequencies by pressing a key connected to a GPIO.

Modifications made

Changed `BOARD_GPIO_KEY_* and BOARD_GPIO_LED_*` in `board.h` to the gpios configured in `gpio_pins.c` in `./examples/fsimx6sx.`

Added

```
configure_gpio_pin(BOARD_GPIO_LED_BLINK_IMX);
```

in `hardware_init.c.`

Added

```
#define BOARD_GPIO_LED_BLINK_IMX        (&gpioBoard_Specific_Entry)
```

in `board.h`

Changed checked value for *GPIO_ReadPinInput()* in *GPIO_Ctrl_WaitKeyPressed()* from 0 to 1.

Changes needed

Connect a KEY and a LED to the following GPIO-pin

efusa9x

Set as	Function	Device	GPIO	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
KEY	I2C_B_IRQ	I2C3_IRQn	GPIO7_IO01	J4_86	J22_48
LED, BLINK	PWM_A	PWM5	GPIO3_IO24	J4_25	J22_32

PicoCOMA9X

Set as	Function	Device	GPIO	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
KEY	SPI_MISO	SPI1	GPIO2_IO11	J2_26	J10_3	J3_3
LED, BLINK	SPI_MOSI	SPI1	GPIO2_IO15	J2_29	J10_6	J3_6

PicoCoreMX6SX

Set as	Function	Device	GPIO	PicoCoreMX6SX Rev 1.10	PCoreBBRGB Rev 1.20
KEY	GPIO4_25	GPIO	GPIO4_IO25	J1_26	J10_4
LED, BLINK	GPIO4_26	GPIO	GPIO4_IO26	J1_24	J10_3

Connect the KEY to 3.3V and the LED to GND.

If you do not want to connect a LED you can change the line in `board.h`.

```
#define BOARD_GPIO_LED_ BLINK_IMX     (&gpioBoard_Specific_Entry)
```

to

```
#undef BOARD_GPIO_LED_ BLINK_IMX
```

Execute binary

Run

```
setenv m4_file "blinking_imx_demo_epit.bin"
run m4
```

9.2.7 can_wakeup

Description

In this demo application you can see the low power management feature of the Cortex-M4 with CAN in stop-mode. The Cortex-M4 will enter stop-mode after the receiver program is running on the Cortex-M4 and the Cortex-A9 is booted up.

It's possible to send Cortex-A9 into suspend mode by entering

```
echo mem > /sys/power/state
```

inside of the Linux system and then start the transmitter program on the other board. This will wake up the Cortex-A9 while the Cortex-M4 is receiving the data sent.

For further information on how to set up the two boards consult the "`Getting_Started_with_FreeRTOS_BSP_for_i.MX_6SoloX.pdf`" located in the `doc/imx6sx` folder in the FreeRTOS BSP package.

Modifications made

Changed `/armgcc/CMakeLists.txt` to set `CMAKE_EXE_LINKER_FLAGS_*` to TCM instead of the default QSPI. This makes it possible to run the program directly on the Cortex-M4 memory. Also removed the shared clocks initialization form `main.c` of the receive part because it's already done at `hardware_init.c`.

Changes needed

Connect CAN_*_H to the corresponding CAN_*_H on the second board and do the same with CAN_*_L.

efusa9x

Function	Device	efusA9X Rev 1.2X	Function	efus-SINTF Rev 1.30
CAN_B_TX	FLEXCAN2	J4_35	CAN_B_H	J22_55
CAN_B_RX	FLEXCAN2	J4_37	CAN_B_L	J22_56

PicoCOMA9X

Function	Device	PicoCOMA9X Rev 1.10	Function	PC2-SINTF	PCOMnet
CAN_A_TX	FLEXCAN1	J2_30	CANH	J10_25	J3_13
CAN_A_RX	FLEXCAN1	J2_31	CANL	J10_26	J3_14

PicoCoreMX6SX

Function	Device	PicoCoreMX6SX Rev 1.10	Function	PCoreBBRGB Rec 1.20
CAN_A_TX	FLEXCAN1	J1_12	CAN_1_H	J6_4
CAN_A_RX	FLEXCAN1	J1_10	CAN_1_L	J6_3

Execute binary

Run

```
mw.w 0x91f000 0x04; dcache flush
```

to clear the shared memory magic numbers and then

```
setenv m4_file "can_wakeup_rx.bin"
run m4; boot
```

on board one. After the Cortex-A9 has booted up, you can execute Listing on board 1 and change to board 2 and run

```
setenv m4_file "can_wakeup_tx.bin"
run m4
```

You should see sent and received packets on both Cortex-M4 Screens and the Cortex-A9 woken up.

9.2.8 fs_adc_i2c_rpmsg_demo

Description

To show the benefits of using an auxiliary real time core in combination with the Linux operating system, F&S created a demo application which consists of two parts:

- The `fs_adc_i2c_rpmsg_demo.bin` which can be executed by the Cortex-M4
- A kernel driver `fs_rpmsg`, which will be automatically loaded while booting the Cortex-A9

The first application will initialize the ADC on the I2C Extension Board from F&S, trigger a conversion request to the ADC and fetching the results back via I2C. Afterwards, these results will be send one by one to the Cortex-A9 via RPMsg.

On the Cortex-A9 side, the driver will fetch the results from the RPMsg buffer and append it to a ring-buffer. It also create a sysfs entry under

/sys/bus/rpmsg/drivers/fs_rpmsg/rpmsg0/buffer

By using cat you're able to retrieve the results on the command line as a comma separated string.

```
# /sys/bus/rpmsg/drivers/fs_rpmsg/rpmsg0/buffer
1.68V, 1.68V, 1.68V, 1.68V, 0.73V, 0.34V, 0.34V
```

By default the driver is deactivated so first you have to activate the kernel driver via `make menuconfig`, compile the kernel and install it on your corresponding board.

Modifications made

These demo was created by F&S.

Changes needed

Connect a potentiometer to a voltage divider, which divides your 3.3 V input voltage down to 2.5 V (the internal reference voltage of the used ADC ADS7828). Then connect it to CH0 on the I2C Extension Board (see table below).

I2C-Extension-Board

Name	Connector
I2C_SCL	J1_11
I2C_SDA	J1_10

Name	Connector
CH0	J2_17
COM	J2_25

FreeRTOS examples

efusa9x

Function	Device	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
I2C_B_SCL	I2C3	J4_84	J22_46
I2C_B_SDA	I2C3	J4_82	J22_45

PicoCOMA9x

Function	Device	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
I2C_A_SCL	I2C4	J2_33	J10_10	J3_10
I2C_A_SDA	I2C4	J2_32	J10_9	J3_9

PicoCoreMX6SX

Function	Device	PicoCoreMX6SX Rev 1.20	PCoreBBRGB Rec 1.20
I2C_A_SCL	I2C4	J1_21	n.c.
I2C_A_SDA	I2C4	J1_23	n.c.

Alternatively you can use an external 3.3V reference voltage by setting the constant `ads7828InitConfig.pd_selection` in `main.c` to `adOn` and flip switch 3 from S2 at the I2C-Extension-Board to *on* as seen in the picture below.

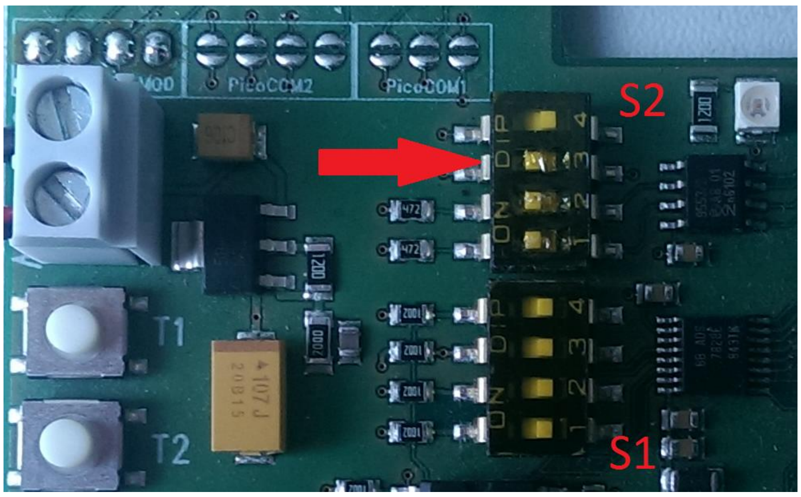


Figure 3: I2C extension board jumpers

Activate the fs_rpmsg driver in `make menuconfig`. Typing “/” will open a search-bar there you can search fs_rpmsg, now select the fs_rpmsg driver by typing the corresponding number which you can see on the left of the block e.g.“1” and then type either “y” for activating it or “n” to deactivate it. After that you can exit the `menuconfig` and save the changes. Make sure to deactivate it later if you want to run other rpmsg examples.

If you are working with the PicoCOMA9X you also need to activate jumper 1 and 2 from S2. This jumpers are setting a Pull-up on the I2C wires because neither the module itself nor the baseboard have Pull-ups on the I2C data and clock wire.

Execute binary

If everything is set up, run

```
setenv m4_file "fs_adc_i2c_rpmsg_demo.bin"

run m4
```

on the Cortex-M4, then boot Linux on the Cortex-A9. After you logged in, you can use

```
cat /sys/bus/rpmsg/drivers/fs_rpmsg/ virtio0.rpmsg-openamp-demo-
channel.-1.30/buffer
```

to get the results.

9.2.9 fs_i2c_extension_board_demo

Description

This demo shows how to use I2C to control devices via master/slave-communication. The demo was designed by F&S to use the I2C Extension Board with FreeRTOS and a PCA9555 driver provided by F&S instead of onboard sensors. You can choose between a polling and interrupt demo on boot up.

Modifications made

A PCA9555 driver was implemented by F&S, the `main.c` was also reworked to communicate with the I2C Extension Board. The key for switching the LEDs has been debounced.

Changes needed

You have to connect the I2C pins on the Starterkit with the corresponding ones on the I2C Extension Board. If you want to run the interrupt demo, you also have to connect a *KEY* to the corresponding GPIO and 3.3V

efusa9x

Function	Device	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
I2C_B_SCL	I2C3	J4_84	J22_46
I2C_B_SDA	I2C3	J4_82	J22_45

Set as	Function	Device	GPIO	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
KEY	I2C_B_IRQ	I2C3_IRQn	GPIO7_IO01	J4_86	J22_48

PicoCOMA9X

Function	Device	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
I2C_A_SCL	I2C4	J2_33	J10_10	J3_10
I2C_A_SDA	I2C4	J2_32	J10_9	J3_9

Set as	Function	Device	GPIO	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
KEY	SPI_MISO	SPI1	GPIO2_IO11	J2_26	J10_3	J3_3

PicoCoreMX6SX

Function	Device	PicoCoreMX6SX Rev 1.20	PCoreBBRGB Rec 1.20
I2C_A_SCL	I2C4	J1_21	n.c.
I2C_A_SDA	I2C4	J1_23	n.c.

Set as	Function	Device	GPIO	PicoCoreMX6SX Rev 1.10	PCoreBBRGB Rev 1.20
KEY	GPIO4_25	GPIO	GPIO4_IO25	J1_26	J10_4

I2C-Extension-Board

Name	Connector
I2C_SCL	J1_11
I2C_SDA	J1_10

Execute binary

Connect the pins on the I2C Extension Board and the Starterkit like stated in the paragraph above and run

```
setenv m4_file "fs_i2c_extension_board_demo.bin"
run m4
```

After the demo launched you will get the following screen:

```
---------- i.MX 6SoloX i2c extension board example -----------



Please select the i2c demo you want to run:
[1].PCA9555 I2C Extension Board Polling Demo
[2].PCA9555 I2C Extension Board Interrupt Demo
---------------------------------------------
```

Now you can choose between a polling and interrupt version by pressing the corresponding key on the keyboard.

If you press `'1'`, you will get

```
---------- i.MX 6SoloX i2c extension board example -----------



Please select the i2c demo you want to run:
[1].PCA9555 I2C Extension Board Polling Demo
[2].PCA9555 I2C Extension Board Interrupt Demo
--------------------------------------------
Your choice: Polling Demo

The LEDs have been successfully turned off!
The LEDs have been successfully turned on!
The LEDs have been successfully turned off!
The LEDs have been successfully turned on!
The LEDs have been successfully turned off!
```

And the LEDs on the I2C Extension board will be toggled every 1 s.

If you press `'2'`, you will get

```
---------- i.MX 6SoloX i2c extension board example -----------



Please select the i2c demo you want to run:
[1].PCA9555 I2C Extension Board Polling Demo
[2].PCA9555 I2C Extension Board Interrupt Demo
--------------------------------------------
Your choice: Interrupt Demo

The LEDs have been successfully turned off!
The LEDs have been successfully turned on!
```

And the LEDs will be toggled if you press the button connected to the `BOARD_GPIO_KEY`.

9.2.10 periodic_wfi_tcm

Description

This demo also highlights the low power management feature of the Cortex-M4 by setting itself in sleep mode, informing Cortex-A9 about its power state who then can shut down all peripherals which are not needed anymore.

Modifications made

Commented out some lines in `main.c` since the demo now uses TCM instead of QSPI.

Changed CMAKE_EXE_LINKER_FLAGS_* to use `*tcm_lowpower.ld` instead of `*qspi2b.ld`

Changes needed

None

Execute binary

Run

```
mw.w 0x91f000 0x0 4; dcache flush
```

to clear the shared memory magic numbers and then

```
setenv m4_file "periodic_wfi.bin"
run m4
```

to kick off the demo. After the Cortex-A9 has booted, you can use the command in listing to set the Cortex-A9 in dormant mode. The Cortex-M4 will reactivate it after the EPIT has triggered.

9.2.11 pingpong_bm

Description

The Master peer on Linux side sends an integer to the Cortex-M4 application, which adds one and transfers it back. This demo works on bare metal base.

Modifications made

Added

```
LMEM_FlushSystemCache(LMEM);
LMEM_InvalidateSystemCache(LMEM);
```

to `middleware/multicore/rpmsg_lite/lib/rpmsg_lite/rpmsg_lite.c`

`Ported the rpmsg_openAMP examples to rpmsg_lite.`

Changes needed

Make sure that the fs_rpmsg driver is deactivated.

Execute binary

First run

```
setenv m4_file "rpmsg_pingpong_bm_example.bin"
run m4; boot
```

then wait for Linux OS to finish booting. Log in, then type

```
modprobe imx_rpmsg_pingpong
```

to load the pingpong master side module. After this you should see

```
Get Data From Master Side : 0
Get Data From Master Side : 2
Get Data From Master Side : 4
```

on Cortex-M4 and

```
get 1 (src: 0x0)
get 3 (src: 0x0)
get 5 (src: 0x0)
```

on Cortex-A9 side.

9.2.12 pingpong_freertos

Description

The Master peer on Linux side sends an integer to the Cortex-M4 application, which adds one and transfers it back. The FreeRTOS RPMsg API is used here.

Modifications made

See Modifications made in **pingpong_bm**.

Changes needed

Make sure that the fs_rpmsg driver is deactivated.

Execute binary

First run

```
setenv m4_file "rpmsg_pingpong_freertos_example.bin"
run m4; boot
```

then wait for Linux OS to finish booting. Log in, then type

```
modprobe imx_rpmsg_pingpong
```

to load the pingpong master side module. After this you should see

```
Get Data From Master Side : 0
Get Data From Master Side : 2
Get Data From Master Side : 4
```

on Cortex-M4 and

```
get 1 (src: 0x0)
get 3 (src: 0x0)
get 5 (src: 0x0)
```

on Cortex-A9 side.

9.2.13 str_echo_bm

Description

This demo demonstrate the RPMsg extension API by creating a channel from Cortex-A9 to Cortex-M4 via `/dev/ttyRPMSG`. After initialization, you can enter a string which then will be replied back and can be read from `/dev/ttyRPMSG`. This demo works on bare metal base.

Modifications made

See Modifications made in **pingpong_bm**.

Changes needed

Make sure that the fs_rpmsg driver is deactivated.

Execute binary

First run

```
setenv m4_file "rpmsg_str_echo_bm_example.bin"
run m4; boot
```

then wait for Linux OS to finish booting. Log in, then type

```
modprobe imx_rpmsg_tty
```

to load the tty master side module.

Now you can echo content via /dev/ttyRPMSG to the Cortex-M4, which then will reply this back to said device:

```
echo 'Test' > /dev/ttyRPMSG30 && read x < /dev/ttyRPMSG30 && echo
$x
```

This will send the string “Test” to the Cortex-M4, read out /dev/ttyRPMSG again to a variable and print this in the terminal.

9.2.14 str_echo_freertos

Description

This demo demonstrate the FreeRTOS RPMsg extension API by creating a channel from Cortex-A9 to Cortex-M4 via `/dev/ttyRPMSG`. After initialization, you can enter a string which then will be replied back and can be read from `/dev/ttyRPMSG`. This demo uses the FreeRTOS RPMsg API.

Modifications made

See Modifications made in **pingpong_bm**.

Changes needed

Make sure that the fs_rpmsg driver is deactivated.

Execute binary

First run

```
setenv m4_file "rpmsg_str_echo_freertos_example.bin"
run m4; boot
```

then wait for Linux OS to finish booting. Log in, then type

```
modprobe imx_rpmsg_tty
```

to load the tty master side module.

Now you can echo content via /dev/ttyRPMSG to the Cortex-M4, which then will reply this back to said device:

```
echo 'Test' > /dev/ttyRPMSG30 && read x < /dev/ttyRPMSG30 && echo
$x
```

This will send the string “Test” to the Cortex-M4, read out `/dev/ttyRPMSG` again to a variable and print this in the terminal.

9.2.15 sema4_demo

Description

Simple demo which shows how to implement a multicore mutex without spinning with CPU.

Modifications made

None

Changes needed

None

Execute binary

Run

```
setenv m4_file "sema4_demo.bin"
run m4
```

to kick off the demo.

9.2.16 sensor_demo

This example wasn't ported because the efusA9X does not provide any onboard sensors.

You can still find the demo in

`not_tested/fsimx6sx/sensor_demo`

but you should use the `i2c_extension_board_demo` with the I2C Extension Board from F&S instead.

9.3 driver_examples

9.3.1 adc_imx6sx (efusa9x only)

Description

This example demonstrate the usage of the ADC on the efusA9X by measuring the AD input on the ADC1_IN0, converting and printing the result every 5 second to the console.

Modifications made

None

Changes needed

The efusA9X does not have the RN3 resistor network (22R) connected on the board by default, so keep in mind to add this if you want to run the example.

Execute binary

Run

```
setenv m4_file "adc_imx6sx_example.bin"
run m4
```

9.3.2 ecspi_interrupt

Description

In this example the master board transfers an array to the slave board, which is then send back to the master. This demo uses interrupts.

Modifications made

Inserted function calls

```
ECSPI_SetDataInactiveState(BOARD_ECSPI_MASTER_BASEADDR,
BOARD_ECSPI_MASTER_CHANNEL, ecspiDataLineStayLow);

and
ECSPI_SetDataInactiveState(BOARD_ECSPI_SLAVE_BASEADDR,
BOARD_ECSPI_SLAVE_CHANNEL, ecspiDataLineStayLow);
```

to the master and slave `main.c` to keep MOSI low between assertion of CS and starting the clock.

Changes needed

You have to connect the masters MOSI, MISO, CLK and CS1 ping with their counterparts on the slave board (MOSI → MOSI, MISO → MISO, CLK → CLK and SSx/CSx → SSx/CSx).

efusa9x

Function	Device	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
SPI_B_MISO	SPI1	J4_50	J22_23
SPI_B_MOSI	SPI1	J4_52	J22_24
SPI_B_CLK	SPI1	J4_54	J22_25
SPI_B_CS1	SPI1	J4_56	J22_26

PicoCOMA9X

Function	Device	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
SPI_MISO	SPI1	J2_26	J10_3	J3_3
SPI_MOSI	SPI1	J2_29	J10_6	J3_6
SPI_CLK	SPI1	J2_28	J10_5	J3_5
SPI_CS0	SPI1	J2_27	J10_4	J3_4

PicoCoreMX6SX

Function	Device	PicoCoreMX6SX Rev 1.20	PCoreBBRGB Rec 1.20
SPI_A_MISO	ECSPI5	J1_16	J10_17
SPI_A_MOSI	ECSPI5	J1_18	J10_16
SPI_A_CLK	ECSPI5	J1_20	J10_12
SPI_A_SS0	ECSPI5	J1_14	J10_14

Execute binary

First run

```
setenv m4_file "ecspi_interrupt_master_example.bin"
run m4
```

on the master board. Wait for the following line to appear on the terminal connected to the Cortex-M4:

```
Press "s" when spi slave is ready.
```

Now connect to the slave board and run

```
setenv m4_file "ecspi_interrupt_slave_example.bin"
run m4
```

If the demo kicks off on the slave board, press “s” on the terminal connected to the masters board Cortex-M4. Now you should see data transmitted between the two boards.

9.3.3 ecspi_polling

Description

In this example the master board transfers an array to the slave board, which is then send back to the master. This demo uses the polling mode for achieving its goal.

Modifications made

Inserted function calls

```
ECSPI_SetDataInactiveState(BOARD_ECSPI_MASTER_BASEADDR,
BOARD_ECSPI_MASTER_CHANNEL, ecspiDataLineStayLow);
```

and

```
ECSPI_SetDataInactiveState(BOARD_ECSPI_SLAVE_BASEADDR,
 BOARD_ECSPI_SLAVE_CHANNEL, ecspiDataLineStayLow);
```

to the master and slave `main.c` to keep MOSI low between assertion of CS and starting the clock.

Changes needed

See **ecspi_interrupt** for information on wiring the two boards.

Execute binary

First run

```
setenv m4_file "ecspi_polling_master_example.bin"
run m4
```

on the master board. Wait for the following line to appear on the terminal connected to the Cortex-M4:

```
Press "s" when spi slave is ready.
```

Now connect to the slave board and run

```
setenv m4_file "ecspi_polling_slave_example.bin"
run m4
```

If the demo kicks off on the slave board, press “s” on the terminal connected to the masters board Cortex-M4. Now you should see data transmitted between the two boards.

Attention

There is a bug inside the iMX6 which prevents the usage of burst lengths greater than [(32 * n) + 1], so don't set it to more than 32!

Note *2*: Maximum Burstlength

9.3.4 epit

Description

Simple application demonstrating two different EPIT instances (EPIT1 and EPIT2) catching each other's counter every 0.5 s.

Modifications made

None

Changes needed

None

Execute binary

Run

```
setenv m4_file "epit_example.bin"
run m4
```

to kick off the demo.

9.3.5 flexcan_loopback_epit

Description

This example demonstrates the FlexCAN module loopback operating mode by sending data from the tx message buffer to its own rx message buffer.

Modifications made

None

Changes needed

None

Execute binary

Run

```
setenv m4_file "flexcan_loopback_epit_example.bin"
run m4
```

9.3.6 flexcan_network_epit

Description

Like in the `can_wakeup` demo this one will send data packets over the CAN bus between two boards.

Modifications made

None

Changes needed

Connect CAN_*_H to the corresponding CAN_*_H on the second board and do the same with CAN_*_L.

efusa9x

Function	Device	efusA9X Rev 1.2X	Function	efus-SINTF Rev 1.30
CAN_B_TX	FLEXCAN2	J4_35	CAN_B_H	J22_55
CAN_B_RX	FLEXCAN2	J4_37	CAN_B_L	J22_56

PicoCOMA9X

Function	Device	PicoCOMA9X Rev 1.10	Function	PC2-SINTF	PCOMnet
CAN_A_TX	FLEXCAN1	J2_30	CANH	J10_25	J3_13
CAN_A_RX	FLEXCAN1	J2_31	CANL	J10_26	J3_14

PicoCoreMX6SX

Function	Device	PicoCoreMX6SX Rev 1.10	Function	PCoreBBRGB Rec 1.20
CAN_A_TX	FLEXCAN1	J1_12	CAN_1_H	J6_4
CAN_A_RX	FLEXCAN1	J1_10	CAN_1_L	J6_3

You need to compile two versions of the software. The first one need

```
#define NODE    1
```

The other must be built with

```
#define NODE    2
```

Save each `*.bin` under a different name (like `flexcan_network_epit_example_b1.bin`) before transferring them to the boards RAM.

Execute binary

Run

```
setenv m4_file "flexcan_network_epit_example_b1.bin"
run m4
```

on board 1 and

```
setenv m4_file "flexcan_network_epit_example_b2.bin"
run m4
```

on the other one.

9.3.7 gpio_imx

Description

This simple application shows how to use LED, buttons, etc. connected to the board via the GPIO interface.

Modifications made

Added

```
configure_gpio_pin(BOARD_GPIO_LED_CONFIG);
```

Changes needed

Connect a *LED* and a *KEY* to the GPIOs.

efusa9x

Set as	Function	Device	GPIO	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
KEY	I2C_B_IRQ	I2C3_IRQn	GPIO7_IO01	J4_86	J22_48
LED, BLINK	PWM_A	PWM5	GPIO3_IO24	J4_25	J22_32

PicoCOMA9X

Set as	Function	Device	GPIO	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
KEY	SPI_MISO	SPI1	GPIO2_IO11	J2_26	J10_3	J3_3
LED, BLINK	SPI_MOSI	SPI1	GPIO2_IO15	J2_29	J10_6	J3_6

PicoCoreMX6SX

Set as	Function	Device	GPIO	PicoCoreMX6SX Rev 1.10	PCoreBBRGB Rev 1.20
KEY	GPIO4_25	GPIO	GPIO4_IO25	J1_26	J10_4
LED, BLINK	GPIO4_26	GPIO	GPIO4_IO26	J1_24	J10_3

Connect the *KEY* to 3.3V and the LED to GND.

Execute binary

Run

```
setenv m4_file "gpio_imx_example.bin"
run m4
```

9.3.8 i2c_interrupt_extension_board_imx6sx

Description

A sample application which uses the FreeRTOS I2C API to let the board communicate as a master with other i2c slaves. It will configure the I2C Extension Board via I2C and then start a chaser light on it to check if the configuration was successful. This application uses interrupts.

Modifications made

Used the `i2c_interrupt_sensor_imx6sx` as a template and changed the purpose and functionality.

Changes needed

You have to connect the I2C pins on the Starterkit with the corresponding ones on the I2C Extension Board.

efusa9x

Function	Device	efusA9X Rev 1.2X	efus-SINTF Rev 1.30
I2C_B_SCL	I2C3	J4_84	J22_46
I2C_B_SDA	I2C3	J4_82	J22_45

PicoCOMA9X

Function	Device	PicoCOMA9X Rev 1.10	PC2-SINTF	PCOMnet
I2C_A_SCL	I2C4	J2_33	J10_10	J3_10
I2C_A_SDA	I2C4	J2_32	J10_9	J3_9

PicoCoreMX6SX

Function	Device	PicoCoreMX6SX Rev 1.20	PCoreBBRGB Rec 1.20
I2C_A_SCL	I2C4	J1_21	n.c.
I2C_A_SDA	I2C4	J1_23	n.c.

I2C-Extension-Board

Name	Connector
I2C_SCL	J1_11
I2C_SDA	J1_10

Execute binary

Connect the pins as stated in “Changes needed”, then run

```
setenv m4_file
"i2c_imx_interrupt_extension_board_imx6sx_example.bin"

run m4
```

to kick of the demo. You will see the following screen and a chaser light on the I2C Extension Board, which goes from the left to the right *LED* and then vice versa:

```
+++++++++++++++ I2C Send/Receive interrupt Example ++++++++++++++

This example will configure the i2c extension board through I2C
Bus

and run a chaser light to see if the i2c extension board was
properly configured.

[1]. Initialize the I2C module with initialize structure.

[2]. Clear input data polarity, so that it will be retained

[3]. Configure Ports as outputs

[4]. Set PCA9555 output port 1 to 0x1

[5]. Start chaser light


Example finished!!!
```

9.3.9 i2c_polling_extension_board_imx6sx

Description

A sample application which uses the FreeRTOS I2C API to let the board communicate as a master with other i2c slaves. It will configure the I2C Extension Board via I2C and then start a chaser light on it to check if the configuration was successful. This application uses polling.

Modifications made

Used the `i2c_interrupt_sensor_imx6sx` as a template and changed the purpose and functionality.

Changes needed

See **i2c_imx_interrupt_extension_board_imx6sx** section `"Changes Needed"` for the required pin connections for the I2C bus.

Execute binary

Connect the pins as stated in "Changes needed", then run

```
setenv m4_file
"i2c_imx_polling_extension_board_imx6sx_example.bin"
run m4
```

to kick of the demo. You will see the following screen and a chaser light on the I2C Extension Board, which goes from the left to the right *LED* and then vice versa:

```
+++++++++++++ I2C Send/Receive polling Example ++++++++++++++++
This example will configure the i2c extension board through I2C
Bus
and run a chaser light to see if the i2c extension board was
properly configured.
[1]. Initialize the I2C module with initialize structure.
[2]. Clear input data polarity, so that it will be retained
[3]. Configure Ports as outputs
[4]. Set PCA9555 output port 1 to 0x1
[5]. Start chaser light
Example finished!!!
```

9.3.10 i2c_interrupt_sensor_imx6sx

This example wasn't ported because the efusA9X does not provide any onboard sensors.

You can still find the demo in

`not_tested/fsimx6sx/sensor_demo`

but you should use the `i2c_interrupt_extension_board_imx6sx` with the I2C Extension Board from F&S instead.

9.3.11 i2c_polling_sensor_imx6sx

See “6.2.8 i2c_interrupt_sensor_imx6sx” for more information.

Use `i2c_polling_extension_board_imx6sx` instead.

9.3.12 uart_polling

Description

This example works similar to the `hello_world` one except that it only echos input and uses a polling interface.

Modifications made

None

Changes needed

None

Execute binary

Run

```
setenv m4_file "uart_imx_polling_example.bin"
run m4
```

9.3.13 uart_interrupt

Description

This example works similar to the `hello_world` one except that it only echos input and uses interrupts.

Modifications made

None

Changes needed

None

Execute binary

Run

```
setenv m4_file "uart_imx_interrupt_example.bin"
run m4
```

9.3.14 wdog_imx

Description

Simple demo which enables WDOG with 1.5s timeout and an interrupt is triggered to refresh the WDOG timer four times.

Modifications made

None

Changes needed

None

Execute binary

Run

```
setenv m4_file "wdog_imx_example.bin"
run m4
```

10 Appendix

List of Figures

List of Tables

Known Issues

As stated in the IMX6SXCE, Rev. 1 from 04/2016, there is an issue with some of the I.MX6 SoloX chips sold on the PicoCOMA9X. This prevents the usage of all RDC_* calls from the FreeRTOS API, otherwise the Cortex-A9 or Cortex-M4 will hang.

This occur on chips with date code less than 1524. Apparently, there is no known solution to this problem. Contact F&S for more information.

See http://cache.freescale.com/files/32bit/doc/errata/IMX6SXCE.pdf?fasp=1&WT_TYPE=Errata&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation&fileExt=.pdf for a detailed description on this.

Third Party Agreement from Real Time Engineers Ltd.

Any FreeRTOS source code, whether modified or in its original release form, or whether in whole or in part, can only be distributed by you under the terms of version 2 of the GNU General Public License plus this exception. An independent module is a module which is not derived from or based on FreeRTOS.

Clause 1: Linking FreeRTOS with other modules is making a combined work based on FreeRTOS. Thus, the terms and conditions of the GNU General Public License V2 cover the whole combination.

As a special exception, the copyright holders of FreeRTOS give you permission to link FreeRTOS with independent modules to produce a statically linked executable, regardless of the license terms of these independent modules, and to copy and distribute the resulting executable under terms of your choice, provided that you also meet, for each linked independent module, the terms and conditions of the license of that module. An independent module is a module which is not derived from or based on FreeRTOS.

Clause 2: FreeRTOS may not be used for any competitive or comparative purpose, including the publication of any form of run time or compile time metric, without the express permission of Real Time Engineers Ltd. (this is the norm within the industry and is intended to ensure information accuracy).

Important Notice

The information in this publication has been carefully checked and is believed to be entirely accurate at the time of publication. F&S Elektronik Systeme assumes no responsibility, however, for possible errors or omissions, or for any consequences resulting from the use of the information contained in this documentation.

F&S Elektronik Systeme reserves the right to make changes in its products or product specifications or product documentation with the intent to improve function or design at any time and without notice and is not required to update this documentation to reflect such changes.

F&S Elektronik Systeme makes no warranty or guarantee regarding the suitability of its products for any particular purpose, nor does F&S Elektronik Systeme assume any liability arising out of the documentation or use of any product and specifically disclaims any and all liability, including without limitation any consequential or incidental damages.

Products are not designed, intended, or authorised for use as components in systems intended for applications intended to support or sustain life, or for any other application in which the failure of the product from F&S Elektronik Systeme could create a situation where personal injury or death may occur. Should the Buyer purchase or use a F&S Elektronik Systeme product for any such unintended or unauthorised application, the Buyer shall indemnify and hold F&S Elektronik Systeme and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, expenses, and reasonable attorney fees arising out of, either directly or indirectly, any claim of personal injury or death that may be associated with such unintended or unauthorised use, even if such claim alleges that F&S Elektronik Systeme was negligent regarding the design or manufacture of said product.