# **FreeRTOS**

FreeRTOS is a popular free real-time operating system widely used with microcontrollers and small microprocessors. FreeRTOS is lightweight and open-source. In addition to real-time applications, FreeRTOS is also suitable for low power, always-on and extremely fast boot applications.



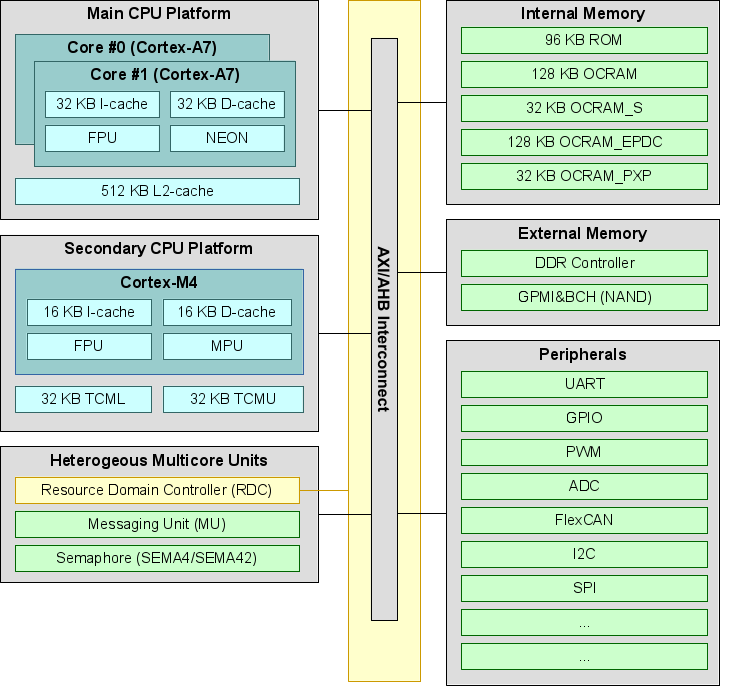
[https://www.freertos.org](https://www.freertos.org/)

# **FreeRTOS on the Cortex-M4 of a Colibri iMX7**

The NXP/Freescale i.MX 7 SoC which is the core of the Colibri iMX7 module implements a heterogeneous asymmetric architecture. Besides the main CPU core(s) based on the ARM Cortex-A7 processor, a secondary general purpose ARM Cortex-M4 core is available too. The secondary core typically runs a RTOS optimized for microcontrollers or a bare-metal application. Toradex provides FreeRTOS, a free professional grade real-time operating system for microcontrollers, along with drivers and several examples which can be used on our Colibri iMX7 platform. The FreeRTOS port is based on NXP FreeRTOS BSP for i.MX 7.

# **Cortex-M4 Overview**

The Cortex-M4 CPU core lives side by side with the Cortex-A7 based primary CPU cores. Both CPU complexes have access to the same interconnect and hence have equally access to all peripherals (shared bus topology). The graphic below is an incomplete and simplified drawing of the architecture with emphasis on the relevant sub systems to understand the heterogeneous asymmetric multicore architecture.



There are several types of memory available. The Cortex-M4 provides local memory (Tightly Coupled Memory, TCM), which is relatively small but can be accessed by the CPU without any latency. There are multiple OCRAM areas (On-Chip RAM, typically SRAM) which are relatively fast as well and slightly larger. The third option is the DDR3 based main memory. From a performance perspective one of the internal areas should be selected whenever possible.

A traditional microcontroller typically has internal NOR flash where the firmware is stored and executed from. This is not the case on Colibri iMX7: There is no NOR flash where the firmware can be flashed onto. Instead, the firmware needs to be stored on the mass storage device such as SD-card or the internal NAND flash. The available mass storage devices are not "memory mapped", and hence application can not be executed directly from any of the cores (no eXecuted-In-Place, XIP). Instead, code need to be loaded into one of the available memory sections before the CPU can start executing it.

The i.MX 7 SoC always boots using the Cortex-A7 core. The core executes the internal boot ROM which typically loads a boot loader such as U-Boot. The boot loader allows loading the firmware from the mass storage device (e.g. NAND flash) into memory, and triggers the Cortex-M4 to start executing the firmware. To upgrade or replace a firmware, one can just replace the firmware binary on the mass storage device.

## **Memory areas**

The two CPU platforms use a different memory layout to access individual sub systems. This table lists some important areas and their memory location for each of the cores side by side. The full list can be found in the i.MX 7 reference manual.



## **Resource Domain Controller**

The RDC prohibits and grants access to peripherals and memory areas for individual bus masters (e.g. CPU, DMA controller) on hardware level. The RDC allows to define up to 4 resource domains, and assign peripherals and memory locations to those resource domains. By default, the A7 core is in domain 0 and all peripherals are assigned to the domain 0. When the FreeRTOS firmware start, the Cortex-M4 core is in domain 0 too, but then reassigns the Cortex-M4 and the required peripherals to domain 1 (see board.c and the example specific hardware\_init.c).

If a device shall be used on the Cortex-M4 which is used by the Linux kernel running on the Cortex-A7 (e.g. I2C), it is important to disable this device in the device tree of the Linux kernel (e.g. set the status property to disabled). The article Device Tree Customization explains in more details how to alter the device tree.

# **Setup Software**

## Update current Linux distribution,

|  |
| --- |
| $ sudo apt update  $ sudo apt upgrade |

## Install 32-bit version of libc and libncurses

|  |
| --- |
| $ sudo dpkg --add-architecture i386  $ sudo apt-get install libc6:i386 libncurses5:i386 |

## Install GNU make and cmake

|  |
| --- |
| $ sudo apt-get install make cmake build-essential git |

## Install Linaro provided ARM Embedded toolchain:

|  |
| --- |
| $ cd ~  $ wget https://launchpad.net/gcc-arm-embedded/4.9/4.9-2015-q3-update/+download/gcc-arm-none-eabi-4\_9-2015q3-20150921-linux.tar.bz2  $ tar xjf gcc-arm-none-eabi-4\_9-2015q3-20150921-linux.tar.bz2 |

## Get the FreeRTOS Source Code

|  |
| --- |
| $ git clone -b colibri-imx7-m4-freertos-v8 git://git.toradex.com/freertos-toradex.git freertos-colibri-imx7/  $ cd freertos-colibri-imx7 |

## Build an example

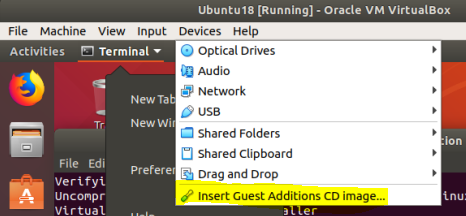
The scripts expect the environment variable to point to the Linaro ARM Embedded toolchain:

|  |
| --- |
| $ export ARMGCC\_DIR=~/gcc-arm-none-eabi-4\_9-2015q3/  $ cd examples/imx7\_colibri\_m4/demo\_apps/hello\_world/armgcc  $ ./build\_all.sh |

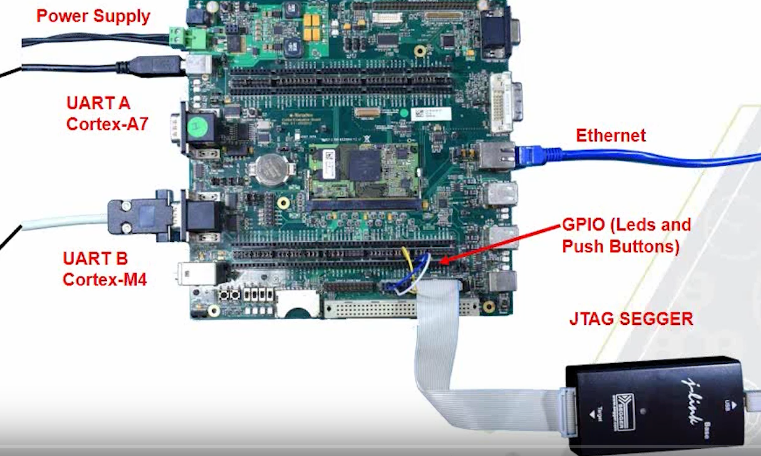
With that, the sub directories release/ and debug/ contain the firmware binaries which can be executed on the target.

**Note on VirtualBox:**

If you are using virtual machine in VirtualBox, be sure to install the guest additions CD:



# **Setup Hardware**



**VGA**

## Connect power supply with 12 V to barrel jack X35 or terminal block connector X33 .

## Connect a VGA monitor to X24.

## Connect an Ethernet cable to X17.

## Connect a USB Type-B to Type-A cable to X27.

## Connect a Serial RS-232 Cable to UART B (Top) X25

1. Connect Segger Jlink to X13
2. Turn on the board.

# **Setup U-Boot**

## Add your user to dialout group

|  |
| --- |
| $ sudo adduser <your\_user\_name> dialout |

## Instal Putty program

|  |
| --- |
| $ sudo apt install putty |

1. Find out the serial port connected to your Linux host.

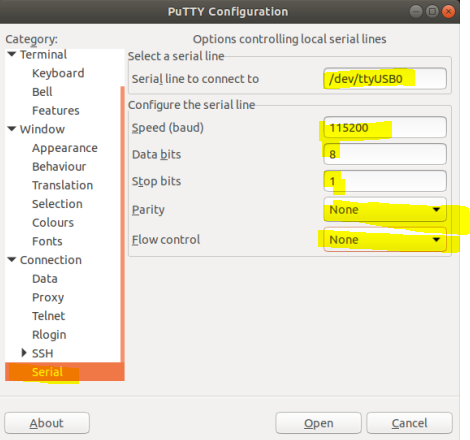
With the Toradex board connected and turned on as in the **Setup Hardware** section, use the next command, the port should be something like “ttyUSB0”

|  |
| --- |
| $ dmesg | grep tty |

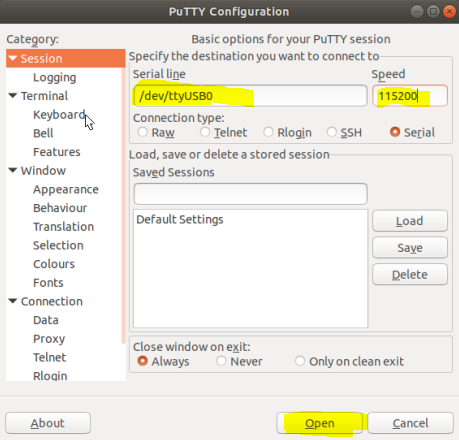
## Open Putty program and configure with the found serial port.

|  |
| --- |
| $ putty |

Select the “Serial” option in the left and make sure the parameters are configured the same, replace the port with the one you found in your machine.

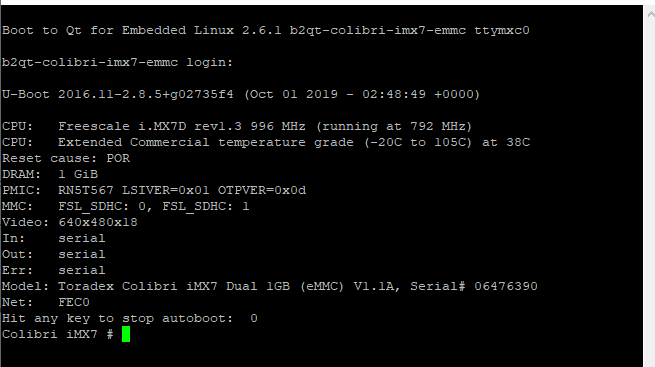


In the main “Session” section, select Serial and press connect:



## Enter U-Boot

The screen will be black, press “Enter” to see the Linux asking for user login, do no enter nothing, reset the board, and quickly in the next screen press any key to enter U-Boot.



## Configure U-Boot with the next commands

|  |
| --- |
| Colibri iMX7 # setenv defargs clk\_ignore\_unused  Colibri iMX7 # setenv fdt\_fixup 'fdt addr ${fdt\_addr\_r} && fdt rm /soc/aips-bus@30800000/spba-bus@30800000/serial@30890000'  Colibri iMX7 # saveenv  Colibri iMX7 # reset |

Linux disables unused clocks by default. However Linux is not aware what clocks are used by the Cortex-M4 core, therefore one should use the clk\_ignore\_unused kernel parameter.

By default, our Linux device tree uses UART\_B too, which leads to a external abort when the Linux kernel tries to access UART\_B. It is recommended to alter the device tree and disable UART\_B using the status property. Temporary, the following fdt\_fixup command can be use in U-Boot

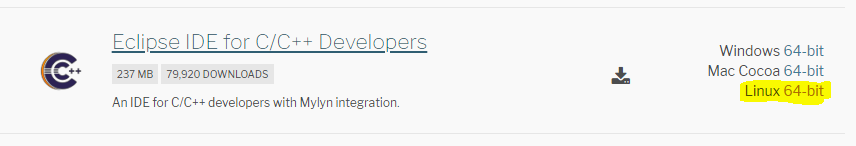
# **Install and Configure Eclipse**

## Install Eclipse

In the Linux host PC download Eclipse from here:

<https://www.eclipse.org/downloads/packages/>

Select the Linux-64-bit version:



Open terminal and move to the path where the file was downloaded and extract it

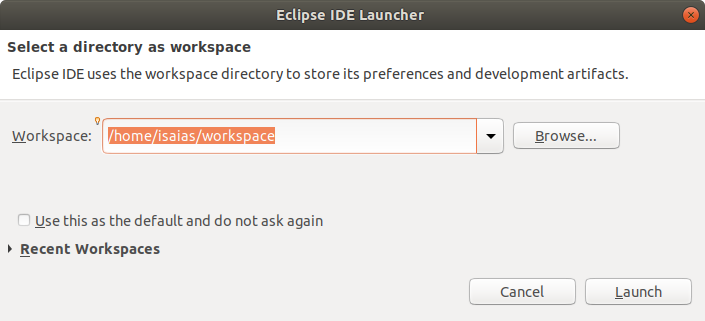
|  |
| --- |
| $ cd ~/Downloads  $ tar -xzf eclipse-cpp-2019-09-R-linux-gtk-x86\_64.tar.gz  $ sudo apt install default-jre |

## Start Eclipse

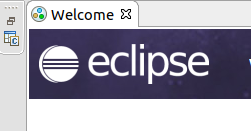
To run eclipse execute:

|  |
| --- |
| $ ~/Downloads/eclipse/eclipse |

Select the workspace and press Launch:

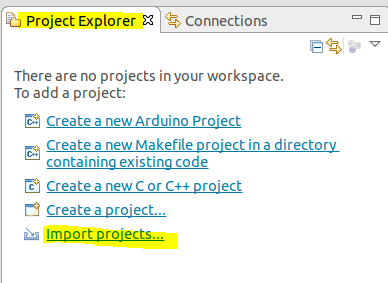


Close Welcome page If open.

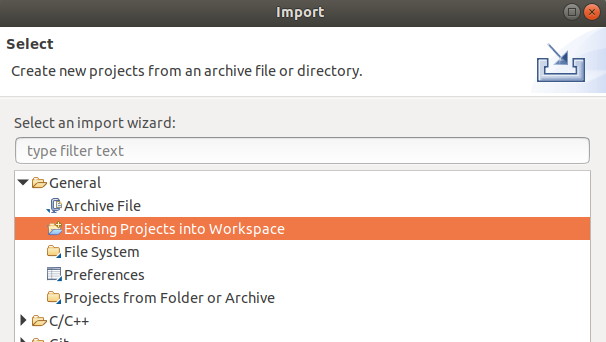


## Import Project

In the “Project Explorer Panel”, import example project

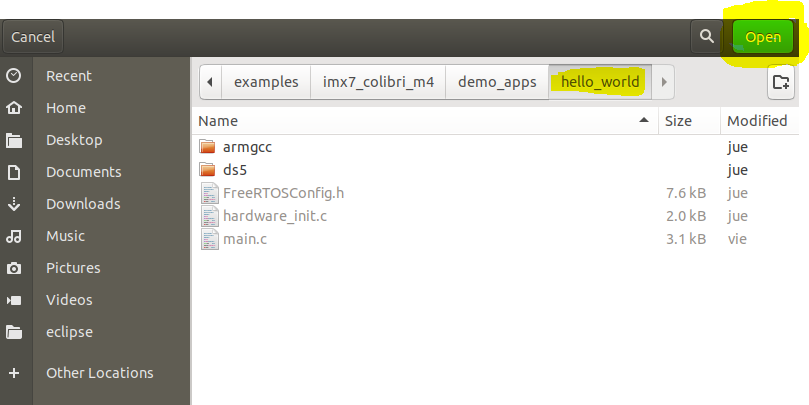


Select “Existing Project”

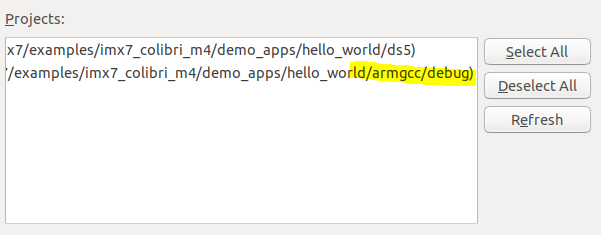


Select the path folder where the hello world program is located and press “Open”

It should be like “freertos-colibri-imx7/examples/imx7\_colibri\_m4/demo\_apps/hello\_world/armgcc”



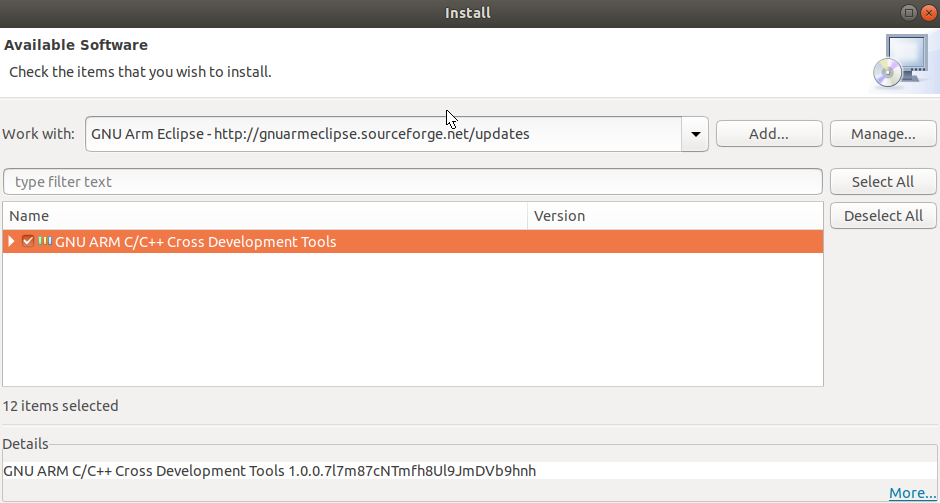
If more than one Project is shown, select only the one that ends with “armgcc/debug”, then press “Finish”



## Install Eclipse Plugin

Help -> Install New Software

Add the URL: <http://gnuarmeclipse.sourceforge.net/updates>



Select and press Next to install and follow the insturctions on screen.

## Install JTAG tools

Download “J-Link Software and Documentation pack for Linux, DEB installer, 64-bit” from:

<https://www.segger.com/downloads/jlink/JLink_Linux_x86_64.deb>

Locate where the file was downloaded, then in a terminal execute

|  |
| --- |
| $ sudo dpkg -i ~/Downloads/JLink\_Linux\_V652e\_x86\_64.deb  $ sudo apt-get install -f |

## Debugger Configuration

Enter to Run -> Debug Configurations..

Select “GDB SEGGER -J-Link Debugging” and click on “New” button

Selecting the new configuration and In the  Tab make sure to have the next configuration:

**J-Link GDB Server Setup** section

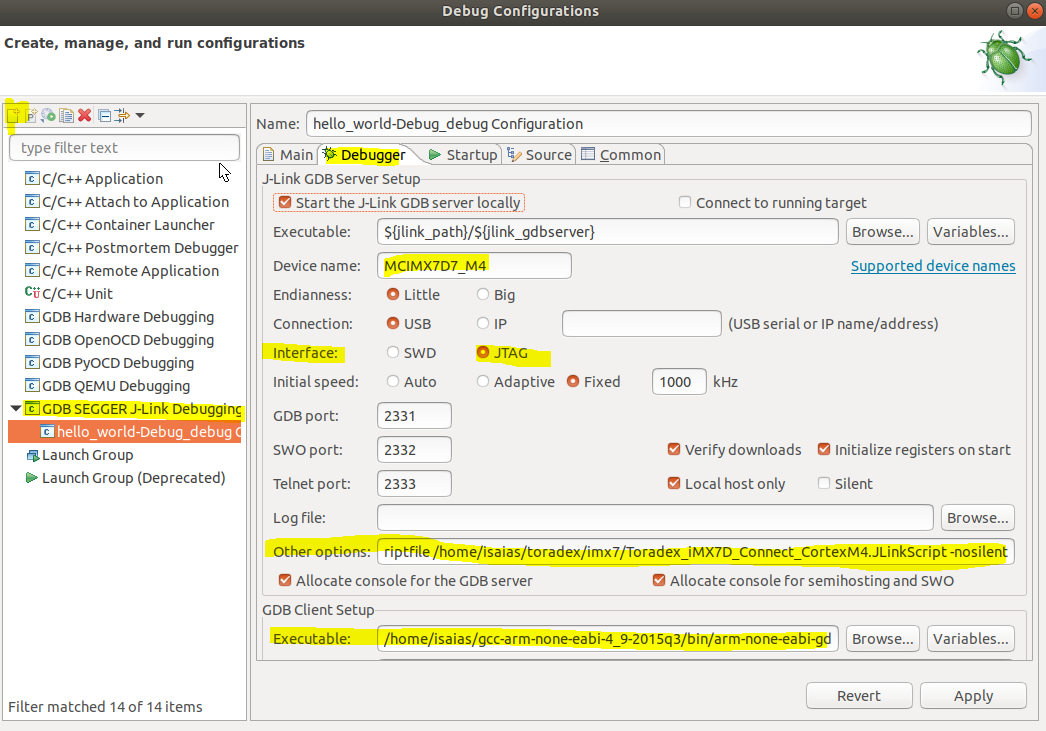
|  |  |
| --- | --- |
| Device name: | MCIMX7D7\_M4 |
| Interface: | JTAG |
| Other options: | -endian little -scriptfile /home/isaias/toradex/imx7/Toradex\_iMX7D\_Connect\_CortexM4.JLinkScript -nosilent |

Download the JLinkScript file from here: [Toradex\_iMX7D\_Connect\_CortexM4.JLinkScript](https://share.toradex.com/6ywfvqe71cm3b45?direct)

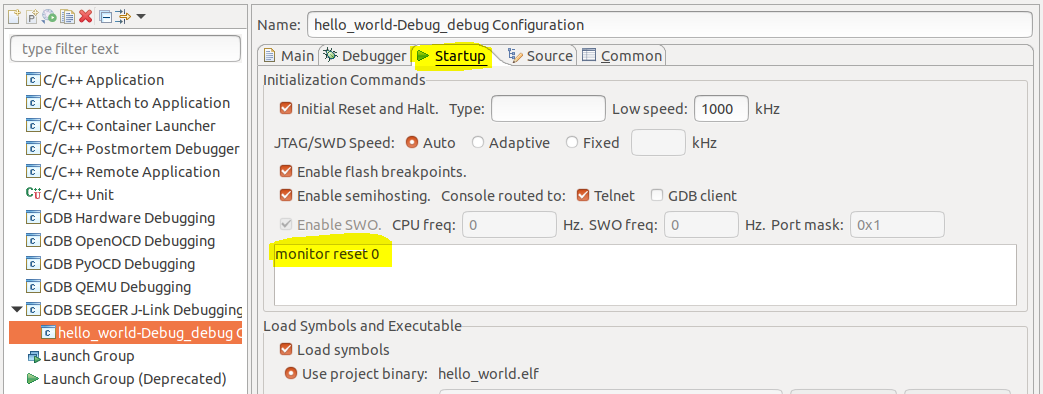
**GDB Client Setup** section

Executable: /home/isaias/gcc-arm-none-eabi-4\_9-2015q3/bin/arm-none-eabi-gdb

**Important:** Be sure to replace the path names, with your corresponding path names.



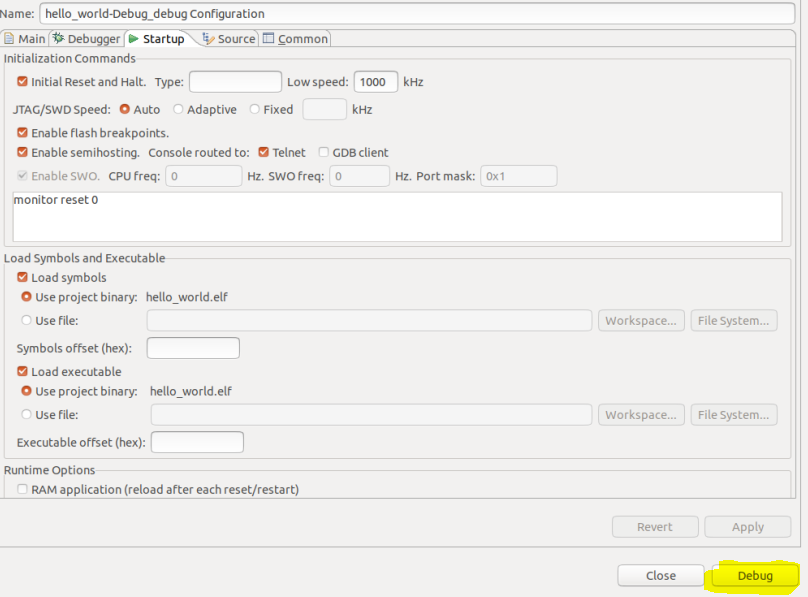
In  Tab, add “monitor reset 0” to the “Initialization Commands”



Finally click on “Apply”

## Debugging Application

Now we are ready to debug the application, press Debug.



# **Running a Firmware on Cortex-M4**

|  |  |
| --- | --- |
| Copy from SD card to memory | fatload mmc 1:1 ${loadaddr} <app\_name>.elf |
| Copy from eMMC to memory | fatload mmc 0:1 ${loadaddr} <app\_name>.elf |
| Copy from tftp to memory | tftp ${loadaddr} <app\_name>.elf |
| Run app | bootaux ${loadaddr} |
| Get dynamic IP address | dhcp |
| Set TFTP server IP address | setenv serverip <ip\_address> |

## **Store a Firmware on Flash and Run it on Boot**

For the Colibri iMX7 1GB, which features eMMC storage, you may just write the M4 firmware to the VFAT partition alongside the Linux kernel and device tree and load it from there.

First on the Colibri Board, create a mount point and mount the first eMMC partition:

|  |
| --- |
| root@colibri-imx7-emmc:/# mkdir /mnt/vfat\_partition  root@colibri-imx7-emmc:/# mount /dev/mmcblk0p1 /mnt/vfat\_partition |

Then, from your Linux host (Virtual Machine), copy your .elf file to that directory:

|  |
| --- |
| $ scp hello\_world.elf root@<board\_ip>:/mnt/vfat\_partition |

Finally, at Colibri Board unmount the partition:

|  |
| --- |
| root@colibri-imx7-emmc:/# umount /mnt/vfat\_partition |

Reset your board, access the U-Boot terminal and update the m4boot environment variable:

|  |
| --- |
| Colibri iMX7 # setenv m4boot 'fatload mmc 0:1 ${loadaddr} hello\_world.elf && bootaux ${loadaddr}'  Colibri iMX7 # saveenv |

For the gpio\_bank2\_imx example, the Switch 1 goes to SODIMM 133 and the Led1 to SODIMM 127

# **Reference**

* [FreeRTOS on the Cortex-M4 of a Colibri iMX7](https://developer.toradex.com/knowledge-base/freertos-on-the-cortex-m4-of-a-colibri-imx7#linux)
* [First steps with Heterogeneous Multicore Processing on the NXP i.MX 7](https://youtu.be/NAIWV70BXKc)
* <https://www.toradex.com/community/questions/18541/can-bus-on-m4-core-imx7.html>