custom embedded devices

eQube board

2CA1004 series

Yocto eGF Developer's guide

Elettronica GF s.r.l. Via Vittori, 63 – 48018 – FAENZA (RA) Tel. (+39) 0546 622245 (r.a.) Fax (+39) 0546 622301
Web: **www.elettronicagf.it** Mail: **sales@elettronicagf.it**

Legal notes

Revision History

Date	Doc. Rev.	Description
09/02/2018	Rev. A01	First Release.
23/02/2018	Rev. A02	Added Kernel, u-boot, Flash SD card and Peripherals section for 0609 series boards

Summary

1 Preface

This document covers the following products:

EVB PCB Code	SOM PCB Code	EVB Part Number
0609	0500	2CA1004

Following guide is applicable to yocto release yocto-egf-0609-001 and later.

2 SDK

Before building the toolchain follow the steps 5.1 and 5.2

Build the toolchain:

```
$ cd yocto\build
$ bitbake meta-toolchain-qt5
```

The output package will be located in tmp/deploy/sdk, run the script to install the toolchain:

```
$ tmp/deploy/sdk/fsl-imx-x11-glibc-x86_64-meta-toolchain-qt5-cortexa9hf-vfp-neon-toolchain-4.1.15-
1.1.0.sh
```

Reply yes to all prompts and do not change the default values.

Every time you need to compile the shell session must be configured:

```
$ source /opt/fsl-imx-x11/4.1.15-1.1.0/environment-setup-cortexa9hf-vfp-neon-egf-linux-gnueabi
```

3 Kernel

3.1 source

Get the kernel source code:

```
$ git clone  https://<user>:<password>@ircost.visualstudio.com/DefaultCollection/CISA%20ePOD/_git/eQUBE_kernel -b
0609_eqube kernel
```

Replace **<user>** and **<password>** with Allegion repositories credentials

3.2 Production kernel

Build:

```
$ ./compile.sh
```

The compile.sh script builds the kernel, the device trees and all the modules. Be aware the compile script overwrites the current .config file with the default config stored in the configs path, so if you have made changes the config, before compiling you must copy it back to the default location:

```
$ cp .config arch/arm/configs/imx_v7_egf_defconfig
```

After the build is terminated the output files will be located in `./build/`***`<kernel-version>`***.
The ***<kernel-version>*** is the value of "CONFIG_LOCALVERSION" variable in the .config file.

3.3 Install

Copy the zImage and device trees to the boot partition of boot device

```
cp build/<kernel-version>/*.dtb /media/sdd1/
cp build/<kernel-version>/zImage /media/sdd1/
```

3.4 Live kernel

Live kernel requires a .cpio.gz rootfs image, the kernel config file points to a live image located in the live_image path into the kernel directory. Remember to copy the .config file back to the default location if you have modified it:

```
$ cp .config arch/arm/configs/imx_v7_egf_update_defconfig
```

Build:

```
$ ./compile.sh update
```

After the build is terminated the output files will be located in `./build/`***`<kernel-version>`***`-live`.

Refer to 'Update on board emmc' section to known how to use the live image.

4 U-boot

4.1 Build

Get the source:

```
$ git clone
https://<user>:<password>@ircost.visualstudio.com/DefaultCollection/CISA%20ePOD/_git/eQUBE_bootloader -b
0609_eqube u-boot
```

Replace **<user>** and **<password>** with Allegion repositories credentials

Build:

```
$ ./compile.sh
```

The compile.sh script builds both u-boot and the SPL binaries.

After the build is finished the output files will be located in `./build/`***`<uboot-version>`*** path, the ***`<uboot-version>`*** is the value of the "EXTRAV" variable in the Makefile. The name of each generated binary contains the version information:

E.g.: assuming EXTRAV=-0609-001 the binary folder will have the following files:

```
SPL-0609-001
u-boot.img-0609-001
```

4.2 Install

The SPL and u-boot binaries must be written to the on-module NOR memory, the programming can be performed from pc or directly on module while the OS is running. Programming from pc is the only way to restore the bootloader if, for any reason, the module fails to boot (corrupted nor, kernel panic ...).

4.2.1 Programming from PC

4.2.1.1 Build production tool

From u-boot folder run the following command:

```
$ ./compile_mfg.sh
```

This script builds one binary for each hardware version.

E.g. If there were three hardware versions (AA0101, AB0101, AC0101) the corresponding binaries would be named as follows:

```
u-boot.imx.wid0500aa0101-0609-001
u-boot.imx.wid0500ab0101-0609-001
u-boot.imx.wid0500ac0101-0609-001
```

The hardware version is always followed by the ***<uboot-version>***, in the above example is 0609-001.

The production binary must be built every time a new version is released, the new binaries will be located along with the old ones.

4.2.1.2 Build programming utility

Get the source:

```
$ git clone https://github.com/elettronicagf/imx-usb-loader.git -b 0609 imx-usb-loader
$ cd imx-usb-loader
```

build:

```
$ make
```

4.2.1.3 Programming

Switch on the board in usb boot mode:

1- Plug the cable to micro usb OTG connector CN1 and connect the board to a pc (usb) running linux
2- Hold pressed the SW2 button
3- Power on the board by pressing SW1 power button
4- Release SW2

from the programming utility folder (imx-usb-loader):

```
$ sudo ./program_spinor.sh <u-boot-version> <hardware-version>
```

Eg. Assuming you want to use u-boot version 0609-001 and hardware version aa0101 run the following command:

```
$ sudo ./program_spinor.sh 0609-001 0500aa0101
```

U-boot and production tool binaries are loaded from u-boot/binaries folder, in the above example the required binaries are:

```
SPL-0609-001
u-boot.img-0609-001
u-boot.imx.wid0500aa0101-0609-001
```

The programming task begins after the command exits, do not power off the board before programming is terminated. Programming status can be observed by looking at the led on the eQube board. During programming LED3,LED4,LED5 and LED6 are powered on. When programming is finished LED3,LED4,LED5 and LED6 are powered off.

4.2.2 Programming from linux

The NOR memory is splitted into partitions:

Device	Name	Size
/dev/mtd0	SPL	256KB

/dev/mtd1	UBOOT	1MB
/dev/mtd2	DATA	768KB

The /dev/mtd2 partition is not used.

Before any write operations the nor must be unlocked, export unlock gpio:
```
echo 90 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio90/direction
```

Unlock
```
echo 1 > /sys/class/gpio/gpio90/value
flash_unlock /dev/mtd0
```

`flash_unlock/flash_lock` commands change locking of all partitions at the same time, thus it's not possible to unlock a single partition.

Write SPL
```
dd if=/dev/mtd0 of=mtdspl.bin bs=1024 count=1
cat ./spl.img >> mtdspl.bin
flashcp mtdspl.bin /dev/mtd0
```

Write U-Boot
```
flashcp u-boot.img /dev/mtd1
```

Lock
```
flash_lock /dev/mtd0
echo 0 > /sys/class/gpio/gpio90/value
```

5 Yocto

5.1 Build host

Assuming you are using Ubuntu, install the required packages:
```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
build-essential chrpath socat libsdl1.2-dev xterm
```

Refer to official Yocto documentation for different build host configuration:
`http://www.yoctoproject.org/docs/2.0.2/mega-manual/mega-manual.html / The build Host Packages`

5.2 Download

Get repo:
```
$ mkdir ~/bin
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
$ export PATH=$PATH:~/bin
```

Prepare directories for yocto sources:
```
$ mkdir ~/yocto
$ cd ~/yocto
$ mkdir sources
$ cd sources
```

Get elettronica GF layer:
```
$ git clone https://<user>:<password>@ircost.visualstudio.com/DefaultCollection/CISA%20ePOD/_git/eQUBE_meta-egf -b 0609_eqube meta-egf
```
Replace **<user>** and **<password>** with Allegion repositories credentials

Get yocto:

```
$ cd ..
$ repo init -u sources/meta-egf -b 0609_eqube -m imx-4.1.15-1.1.2.xml
$ repo sync
```

Setup yocto:

```
$ cd sources
$ cd meta-egf
$ ./scripts/setup-egf.sh
```

This step must be performed only once.

The branch name is 0609_eqube.

5.3 Setup

The meta-egf layer provides the following targets:

Machines:

`0609equbeimx6q`	0609 eQube board imx6 quad/dual

Images:

`egf-image`	Image with X11 without QT5
`egf-image-qt5`	Image with X11, QT5 and all demos
`egf-image-update`	Live image used to update the system

Distro:

`fsl-imx-x11`	reference distro for elettronica GF hardware

SDK:

`meta-toolchain-qt5`	SDK with QT5 support
`meta-toolchain`	SDK without QT support

Setup build environment:

```
$ cd ~/yocto
$ DISTRO=fsl-imx-x11 MACHINE=0609equbeimx6q source ./egf-setup-release.sh –b build
```

You can create different build configuration by changing the build directory after the '`–b`' parameter.

5.4 Build

```
$ bitbake <image>
```

Check the above table for available targets.

5.4.1 Binaries

Images

Build output is located in tmp/deploy/images/0609equbeimx6q, the files are:

`<image>-0609equbeimx6q.sdcard`	This is a complete image ready to be flashed to an sdcard. This image contains 2 partitions, the first

	contains the kernel and device trees, the second contains the rootfs
`<image>-0609equbeimx6q.tar.bz2`	This file is the tarball of the rootfs
`<image>-0609equbeimx6q.cpio.gz`	Live image to be embedded in the kernel, available only for egf-image-update
`zImage`	Kernel image
`zImage-imx6-egf-WID0500_<hw_version>.dtb`	Device tree of a specific hardware configuration. There will be one file for every hardware version of the module.

Eg. If image is "egf-image" and machine is "0609equbeimx6q " the .sdcard filename will be egf-image-0609equbeimx6q .sdcard. Since the module of eQube board is the 0500 and this one has three hw configurations you will find also these files:

```
zImage-imx6-egf-WID0500_AA01.01.dtb
zImage-imx6-egf-WID0500_AB01.01.dtb
zImage-imx6-egf-WID0500_AC01.01.dtb
```

Sdk

The output path is located in tmp/deploy/sdk

`fsl-imx-x11-glibc-x86_64-meta-toolchain-qt5-cortexa9hf-vfp-neon-toolchain-4.1.15-1.1.0.sh`	Machine: x86_64 Distro: fsl-imx-x11 Kernel: 4.1.15 Yocto: 1.1.0

```
SDK name may vary depending on host machine, selected distro, kernel version and yocto version.
```

6 Flash eMMC

6.1 Update onboard eMMC using USB key

U-boot, before loading the production kernel, checks for an update key plugged in and if it finds any tries to boot the system using the kernel and rootfs stored in the key. The USB key can update SPL, u-boot, kernel and rootfs. The script create-update.sh builds the USB key image.

Get the script:

```
git clone https://github.com/elettronicagf/util-update.git -b 0609
```

before running the script the right binaries must be copied to the locations pointed by the script:
the following path are located under the ./binaries path:

`u-boot`	u-boot and spl binaries:
	`u-boot.img-<version>`
	`SPL-<version>`

`kernel/<version>`	kernel, modules, device trees binaries
	`zImage`
	`*.dtb`
	`modules_<version>.tgz`

`kernel/<version>-live`	Live kernel image and device trees (used only for the update process)
	`zImage`
	`*.dtb`

`rootfs`	rootfs image generated by yocto `<rootfsname>-<version>.tar.bz2`
`app`	custom/application files these files are installed to the /home/root folder, the script places the kernel modules here

The script must be fixed with the correct versions, this is done by updating the following variables inside the script (eg.):

```
UBOOT_VERSION=0609-001
SPL_VERSION=$UBOOT_VERSION
KERNEL_VERSION=0609-001
ROOTFS_VERSION=1.0
ROOTFSLIVE_VERSION=0609-001
```

Example of the binaries needed for the above configuration:

```
./u-boot/u-boot.img-0609-001
./u-boot/SPL-0609-001

./rootfs/0609equbeimx6q-1.0.tar.bz2
```

The kernel modules are placed here by the script:

```
./app/home/root/modules/wlcore_sdio.ko
./app/home/root/modules/mac80211.ko
./app/home/root/modules/wlcore.ko
./app/home/root/modules/wl12xx.ko
./app/home/root/modules/cfg80211.ko

./kernel/0609-001/zImage

./kernel/0609-001/imx6-egf-WID0500_AC01.01.dtb
./kernel/0609-001/imx6-egf-WID0500_AB01.01.dtb
./kernel/0609-001/imx6-egf-WID0500_AA01.01.dtb
./kernel/0609-001/modules_0609-001.tgz
```

The script has some commandline options that help the user customize the update process:

```
--no-uboot          do not update uboot
--no-spl            do not update spl
--no-kernel         do not update the kernel
--no-rootfs         do not update the rootfs
--makepartition     force the script to remove and recreate the emmc partitions
```

0609 eQube Board

Plug the usb key in the usb connector CN5 then enable the board power supply.

7 Peripherals

7.1 Temperature Sensors

7.1.1 CPU Temperature Sensor

CPU temperature sensor can be read from file:

/sys/devices/virtual/hwmon/hwmon1/temp1_input

Temperature is reported in m°C.

To protect CPU HW, linux kernel forces a system shutdown when die temperature reaches 105°C.

7.1.2 Board Temperature Sensor

Board temperature sensor can be read from file:

/sys/bus/i2c/drivers/tmp102/0-0048/hwmon/hwmon0/temp1_input

Temperature is reported in m°C.

Suggested battery discharge temperature is 0 - 60°C. It is suggested to power off system when Board Temperature sensor reading is without 0 - 60°C range.

7.1.3 Battery Temperature Sensor for Charge

Battery temperature monitoring during charge is done via HW and it is not readable by SW. Battery charging is enabled only when battery temperature is between 0 and 45°C.

7.2 Power Supply and Battery Charger

7.2.1 Power Present

SW can read whether external power adapter is present or not. This reflects the state of power present LED DL2 (green).

Power present GPIO is mapped to pin 110 in mraa lib:

```
mraa-gpio get 110
```

The returned value is 0 if external power adapter is connected, 1 otherwise.

7.2.2 Charge Indicator

SW can read whether battery is charging or if charge is terminated. This reflects the state of charge LED DL1 (orange).

Charge status GPIO is mapped to pin 111 in mraa lib:

```
mraa-gpio get 111
```

The returned value is 0 if battery is charging, 1 otherwise.

7.2.3 Battery Charge Control

As default setting battery charge is always enable, SW can disable battery charging.

Battery Charge Control GPIO is mapped to pin 112 in mraa lib.

To disable battery charging:

```
mraa-gpio set 112 1
```

To enable battery charging:

```
mraa-gpio set 112 0
```

7.2.4 Battery Charge Status

Battery voltage can be read through using provided mraa lib. Battery ADC is mapped to ADC 0 in mraa library. Following is an example to read battery ADC in mV:

```
/* AIO port */
#define AIO_PORT 0

int main()
{
    mraa_result_t status = MRAA_SUCCESS;
    mraa_aio_context aio;
    uint16_t value = 0;
    float battery_voltage;

    /* initialize mraa for the platform (not needed most of the times) */
    mraa_init();

    /* initialize AIO */
    aio = mraa_aio_init(AIO_PORT);
    if (aio == NULL) {
        fprintf(stderr, "Failed to initialize AIO\n");
        mraa_deinit();
        return EXIT_FAILURE;
    }

    value = mraa_aio_read(aio);
```

```
        battery_voltage = ((float)value * 3300 * 4);
        battery_voltage = battery_voltage / (1024 * 3);
        fprintf(stdout, "Battery voltage is %.0f\n", battery_voltage);

        /* close AIO */
        status = mraa_aio_close(aio);
        if (status != MRAA_SUCCESS) {
            goto err_exit;
        }

        /* deinitialize mraa for the platform (not needed most of the times) */
        mraa_deinit();

        return EXIT_SUCCESS;

    err_exit:
        mraa_result_print(status);

        /* deinitialize mraa for the platform (not needed most of the times) */
        mraa_deinit();

        return EXIT_FAILURE;
    }
```

By characterizing battery discharge curve it is suggested to split battery charge into the following 5 areas:

1) Vbat ≥ 3927 – Battery is fully charged
2) 3927 > Vbat ≥ 3774
3) 3774 > Vbat ≥ 3730
4) 3730 > Vbat > 3600
5) Vbat ≤ 3600, Battery is low, it is suggested to apply policies to protect the system from unwanted shutdowns, for example remount data partitions as read only or start a controlled shutdown.

7.3 Bluetooth – WL18xx

7.3.1 Files

The firmware and calibration files are located in ./firmware/ti-connectivity path. The bt driver loads all the parameters from `/lib/firmware/ti-connectivity/TIInit_11.8.32.bts`, which is integrated in yocto rootfs.

7.3.2 Setup

Switch on the module:

```
mraa-gpio set 106 1
```

Setup hci device:

```
hciattach /dev/ttymxc2 texas
hciconfig hci0 up
```

Test connection:

```
hcitool scan
```

7.4 Bluetooth Silicon Labs BGM111 or BGM13P

Silicon Labs BGM module is controllable through mraa-library. The following is the list of the relevant signals:

- Reset signal is mraa GPIO pin 104. To reset module set GPIO pin to 1, to release reset set GPIO to 0.
- BLE module power enable is mraa GPIO pin 105. To enable power supply for BLE module set GPIO to 1, otherwise set to 0.
- PD14 signal of BLE module is connected to mraa GPIO pin 115.

- PD15 signal of BLE module is connected to mraa GPIO pin 116.

The following is the recommended procedure for module power on:

```
mraa-gpio set 105 1
mraa-gpio set 104 0
```

Module is connected to i.MX6 /dev/ttymxc1 UART which is also mapped to mraa uart dev 1.

7.5 WiFi - WL-18xx

The firmware binaries are located in the root's home (/home/root):
./firmware/ti-connectivity

Since the kernel looks for firmware files in the /lib/firmware path in order to move the files in a different location we need a symbolic link:

```
/lib/firmware/ti-connectivity -> /home/root/firmware/ti-connectivity
```

The kernel is configured to compile the wifi drivers as modules. You can place the modules where ever you like, for convenience we have placed them in the `/home/root/modules` path.
To load wifi driver proceed as follows:

```
cd /home/root/modules
insmod compat.ko
insmod cfg80211.ko
insmod mac80211.ko
insmod wlcore_sdio.ko
insmod wlcore.ko
insmod wl18xx.ko
ifup wlan0
```

7.6 User Leds

User LEDs are managed through mraa library.
Led mapping is the following:
LED3 is "LED3cb"
LED4 is "LED4cb"
LED5 is "LED5cb"
LED6 is "LED6cb".

Following is an example for driving a led through mraa library:

```
#define LEDNAME "LED3cb"
#define LEDTRIGGERNONE "none"

int main(void)
{
    mraa_result_t status = MRAA_SUCCESS;
    mraa_led_context led;
    int val;

    /* initialize mraa for the platform (not needed most of the time) */
    mraa_init();

    /* initialize LED */
    led = mraa_led_init(LEDNAME);
    if (led == NULL) {
        fprintf(stderr, "Failed to initialize LED\n");
        mraa_deinit();
```

```
            return EXIT_FAILURE;
        }

        /* set LED trigger to none */
        status = mraa_led_set_trigger(led, LEDTRIGGERNONE);
        if (status != MRAA_SUCCESS) {
            fprintf(stderr, "unable to set LED trigger\n");
            goto err_exit;
        }

           /* power on led */
           status = mraa_led_set_brightness(led, 1);
        if (status != MRAA_SUCCESS) {
            fprintf(stderr, "unable to set LED Brightness\n");
            goto err_exit;
        }

        usleep(1000000);

        /* power off led */
           status = mraa_led_set_brightness(led, 0);
        if (status != MRAA_SUCCESS) {
            fprintf(stderr, "unable to set LED Brightness\n");
            goto err_exit;
        }

        /* close LED */
        mraa_led_close(led);

        /* deinitialize mraa for the platform (not needed most of the times) */
        mraa_deinit();

        return EXIT_SUCCESS;

    err_exit:
        mraa_result_print(status);

        /* deinitialize mraa for the platform (not needed most of the times) */
        mraa_deinit();

        return EXIT_FAILURE;
    }
```

Leds can be controlled also by trigger. For example trigger “timer” can be used for set 1sec led blinking mode or trigger “heartbeat” can be used to reflect cpu usage.
Following is an example for led driving using triggers:

```
    #define LEDNAME "LED3cb"
    #define LEDTRIGGERTIMER "timer"

    int main(void)
    {
        mraa_result_t status = MRAA_SUCCESS;
        mraa_led_context led;

        /* initialize mraa for the platform (not needed most of the time) */
        mraa_init();

        /* initialize LED */
        led = mraa_led_init(LEDNAME);
        if (led == NULL) {
            fprintf(stderr, "Failed to initialize LED\n");
            mraa_deinit();
            return EXIT_FAILURE;
        }

        /* set LED trigger to none */
        status = mraa_led_set_trigger(led, LEDTRIGGERTIMER);
        if (status != MRAA_SUCCESS) {
            fprintf(stderr, "unable to set LED trigger\n");
            goto err_exit;
        }

        /* close LED */
        mraa_led_close(led);

        /* deinitialize mraa for the platform (not needed most of the times) */
        mraa_deinit();

        return EXIT_SUCCESS;

    err_exit:
        mraa_result_print(status);

        /* deinitialize mraa for the platform (not needed most of the times) */
        mraa_deinit();

        return EXIT_FAILURE;
    }
```

7.7 Buzzer

Buzzer is managed through mraa library. Buzzer is connected to mraa pwm pin 117. Following is an example for driving buzzer through mraa:

```
#define PWM 117
#define PWM_PERIOD_US 340

int main(void)
{
    mraa_result_t status = MRAA_SUCCESS;
    mraa_pwm_context pwm;

    /* initialize mraa for the platform*/
    mraa_init();

    /* Init PWM */
    pwm = mraa_pwm_init(PWM);
    if (pwm == NULL) {
        fprintf(stderr, "Failed to initialize PWM\n");
        mraa_deinit();
        return EXIT_FAILURE;
    }

    /* write PWM duty cyle */
    status = mraa_pwm_write(pwm, 0.5f);
    if (status != MRAA_SUCCESS) {
        goto err_exit;
    }

    /* set PWM period */
    status = mraa_pwm_period_us(pwm, PWM_PERIOD_US);
    if (status != MRAA_SUCCESS) {
        goto err_exit;
    }

    /* enable PWM */
    status = mraa_pwm_enable(pwm, 1);
    if (status != MRAA_SUCCESS) {
        goto err_exit;
    }

    usleep(100000);

    /* disable PWM */
    status = mraa_pwm_enable(pwm, 0);
    if (status != MRAA_SUCCESS) {
        goto err_exit;
    }

    /* close PWM */
    mraa_pwm_close(pwm);

    /* deinitialize mraa for the platform (not needed most of the times) */
    mraa_deinit();

    return EXIT_SUCCESS;

err_exit:
    mraa_result_print(status);

    /* close PWM */
    mraa_pwm_close(pwm);

    /* deinitialize mraa for the platform (not needed most of the times) */
    mraa_deinit();

    return EXIT_FAILURE;
}
```

7.8 Buttons

7.8.1 Power Button

Power button is mapped to pin 109 in mraa library.

```
mraa-gpio get 109
```

Returned value is 0 when button is pressed, 1 if button is released.

7.8.2 Function Button

Function button is mapped to pin 114 in mraa library.

```
mraa-gpio get 114
```

Returned value is 0 when button is pressed, 1 if button is released.

7.8.3 Test Button

Function button is mapped to pin 113 in mraa library.

```
mraa-gpio get 113
```

Returned value is 0 when button is pressed, 1 if button is released.

7.9 Expansion connector CN4 pin mapping

Pin 3 – SOM_ECSPI3_SS0	Configured as GPIO mapped to mraa GPIO pin 3
Pin 5 – SOM_ECSPI3_SCLK	Configured as GPIO mapped to mraa GPIO pin 5
Pin 6 - SOM_ECSPI3_SS1	Configured as GPIO mapped to mraa GPIO pin 6
Pin 7 - SOM_ECSPI3_MISO	Configured as GPIO mapped to mraa GPIO pin 7
Pin 8 - SOM_ECSPI3_MISO	Configured as GPIO mapped to mraa GPIO pin 8
Pin 9 – UART1-RX	Configured as UART (/dev/ttymxc0) and mapped to mraa UART dev 0
Pin 10 – UART1-TX	
Pin 11 – UART1-RTS	
Pin 12 – UART1-CTS	
Pin 13 - SOM_UART5-RX_GPIO4-IO09	Configured as GPIO mapped to mraa GPIO pin 13
Pin 14 - SOM_UART5-TX_GPIO4-IO08	Configured as GPIO mapped to mraa GPIO pin 14
Pin 15 - SOM_UART5-RTS_GPIO4-IO14	Configured as GPIO mapped to mraa GPIO pin 15
Pin 16 - SOM_UART5-CTS_GPIO4-IO15	Configured as GPIO mapped to mraa GPIO pin 16
Pin 17 – I2C3_SCL	Configured as I2C and mapped to mraa I2C bus 0
Pin 18 – LED_GPIO5-IO21	Configured as GPIO mapped to mraa GPIO pin 18
Pin 19 - I2C3_SDA_NVCC	Configured as I2C and mapped to mraa I2C bus 0
Pin 20 - GPIO6-IO00	Configured as GPIO mapped to mraa GPIO pin 20