Linux to Kw40z I2C

In this tutorial, we will connect the beaglebone black and the raspberry pi 3 I2C to the Kinetics kw40z microcontroller via I2C.

The beaglebone or the raspberry pi 3 will be the I2C master and the kw40z will be the slave.

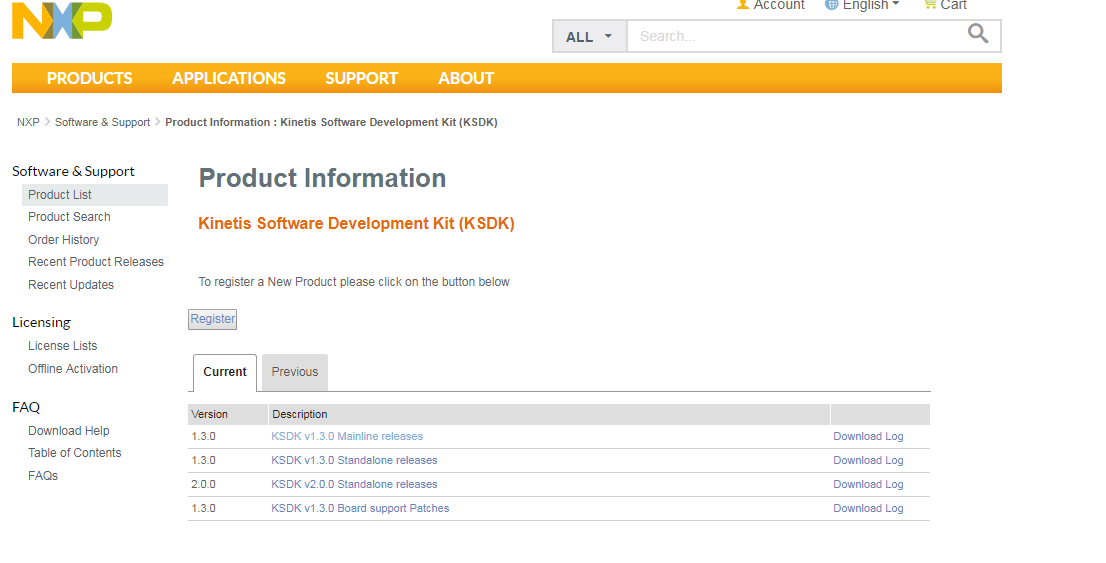
For the I2C slave we will use the kw40z examples for i2c slave.

You need the kinetics sdk 1.3 which support the kw40z

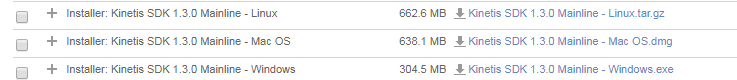
The sdk can be download from nxp web site.

You will need an account in nxp to download

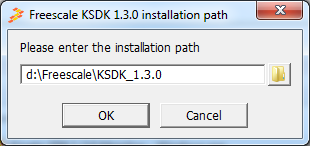
Eventually you will need to reach to this place where you can download version 1.3



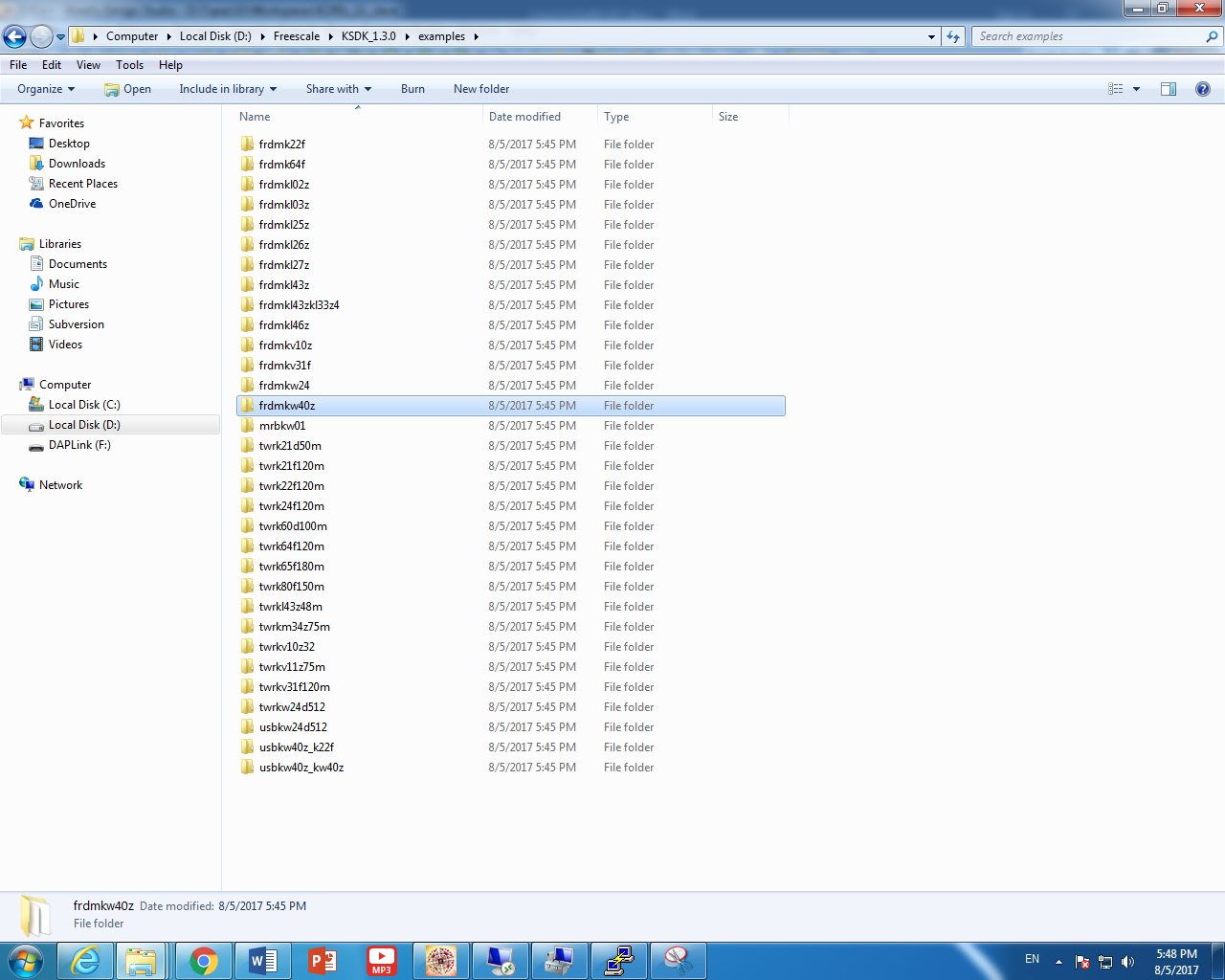
It supports Linux, windows and Mac OS, I am using windows here:



File size is 304 mega and we need to install it

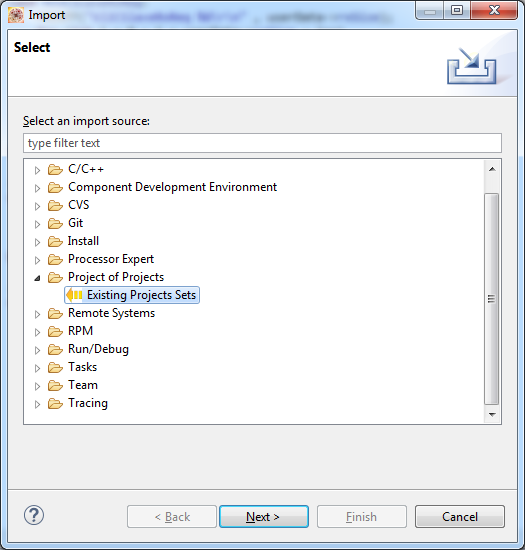


Select the frdkkw40z



D:\Freescale\KSDK\_1.3.0\examples\frdmkw40z\driver\_examples\i2c\i2c\_callback\slave

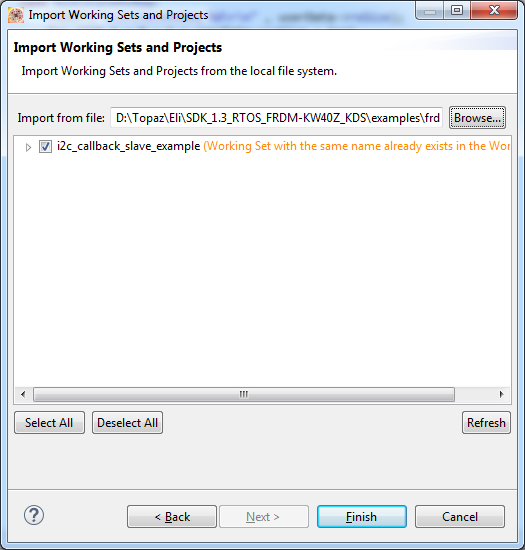
Select the i2c example as project of project set:



SDK\_1.3\_RTOS\_FRDM-KW40Z\_KDS\examples\frdmkw40z\driver\_examples\i2c\i2c\_callback\slave\kds

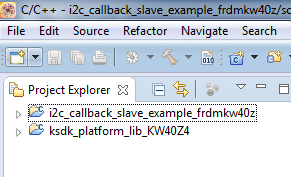
Note: the SDK I am using is the Kinetis SDk 1.3 for the kw40z

Select the kds project.



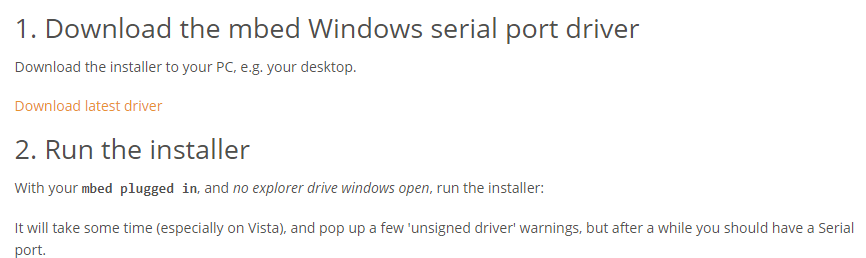
Press finished

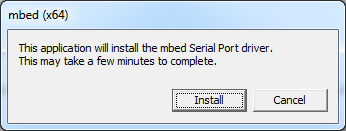
You should see the following project layout:

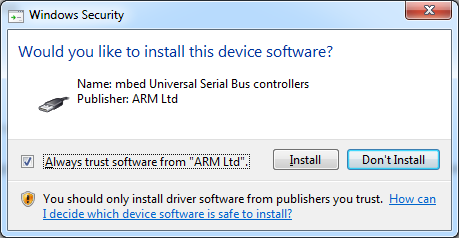


Download the windows mbed driver from:

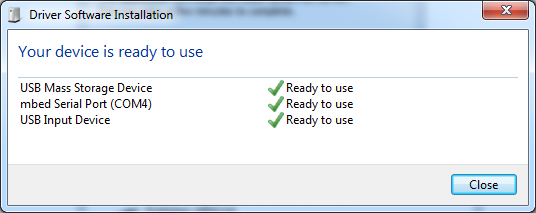
<https://developer.mbed.org/handbook/Windows-serial-configuration>







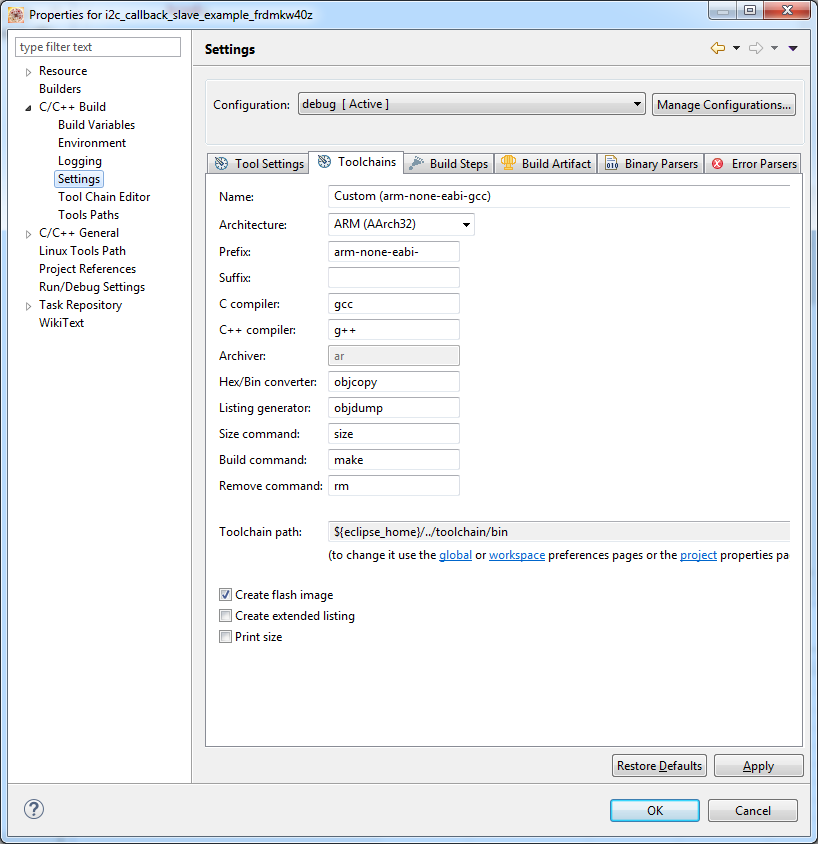
Windows should install the **mbed** serial port driver

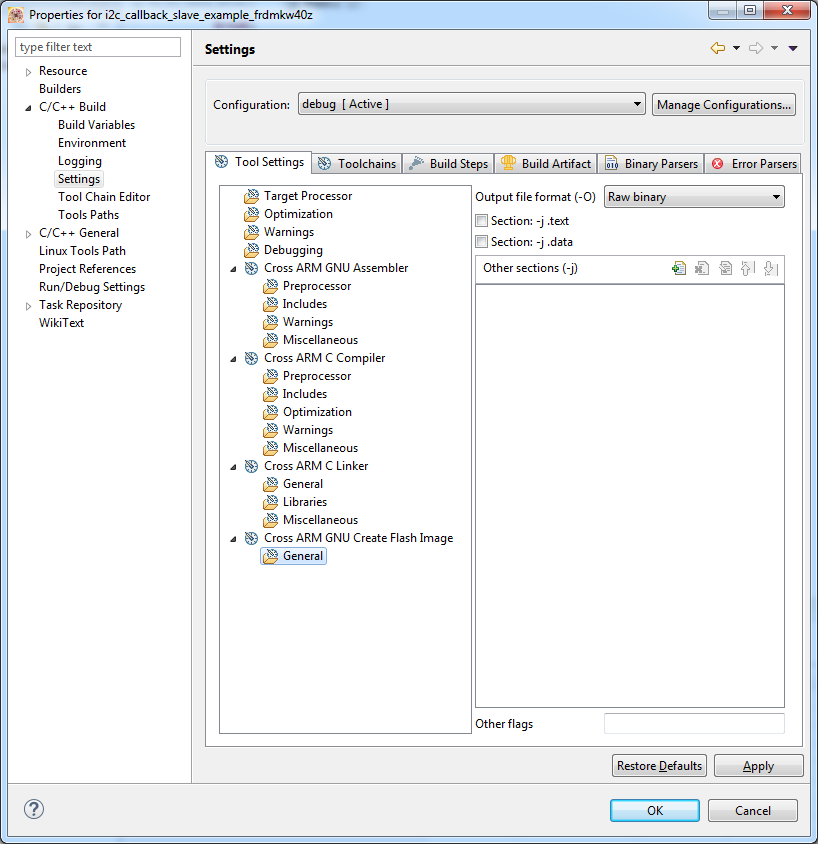


Change the settings in the kinetis design studio to create bin file:

We need the create flash image

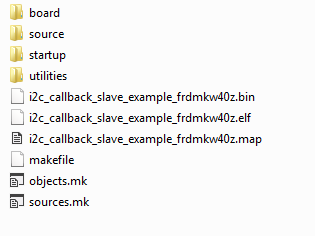
And row binary





We can see in the kds\debug directory a bin file:

<http://www.nxp.com/products/developer-resources/run-time-software/kinetis-software-and-tools/ides-for-kinetis-mcus/opensda-serial-and-debug-adapter:OPENSDA#USB-KW40Z>

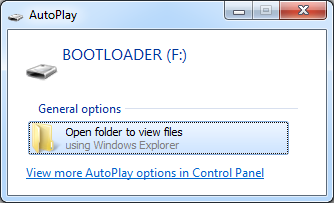


http://www.nxp.com/products/developer-resources/run-time-software/kinetis-software-and-tools/ides-for-kinetis-mcus/opensda-serial-and-debug-adapter:OPENSDA#USB-KW40Z

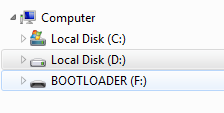
To replace the bootloader do the following

1. Disconnect the usb power cable
2. Hold down the reset button
3. While holding it down connect the power

You should see that the disk is now labled as:



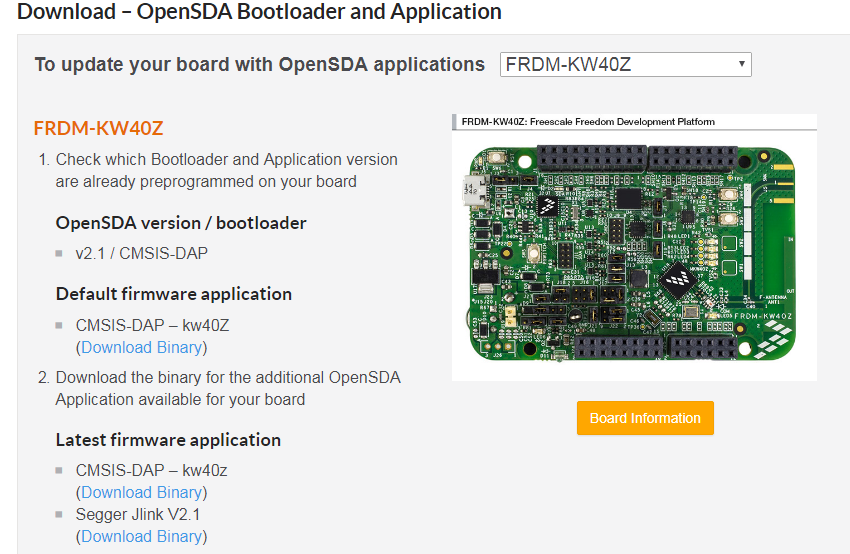
The disk letter can be from D to any free letter.



In my computer, I have two physical hard drives.

Now, drag and drop the k22fn512\_usb\_kw40z\_cmsis-dap.bin into the F drive

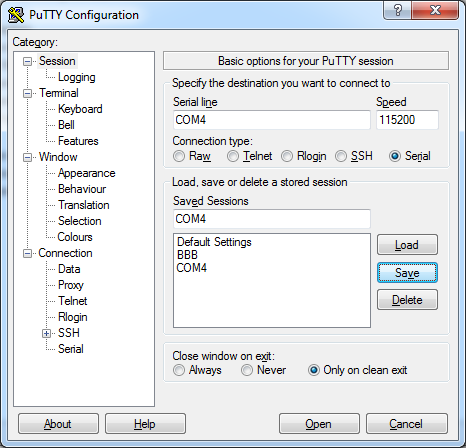
Next , unplug and plug again the usb power cable in order to load the firmware as usual.

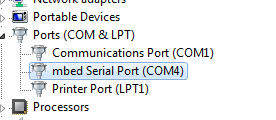


File name is:

k20dx128\_frdm\_kw40z\_cmsis-dap.bin

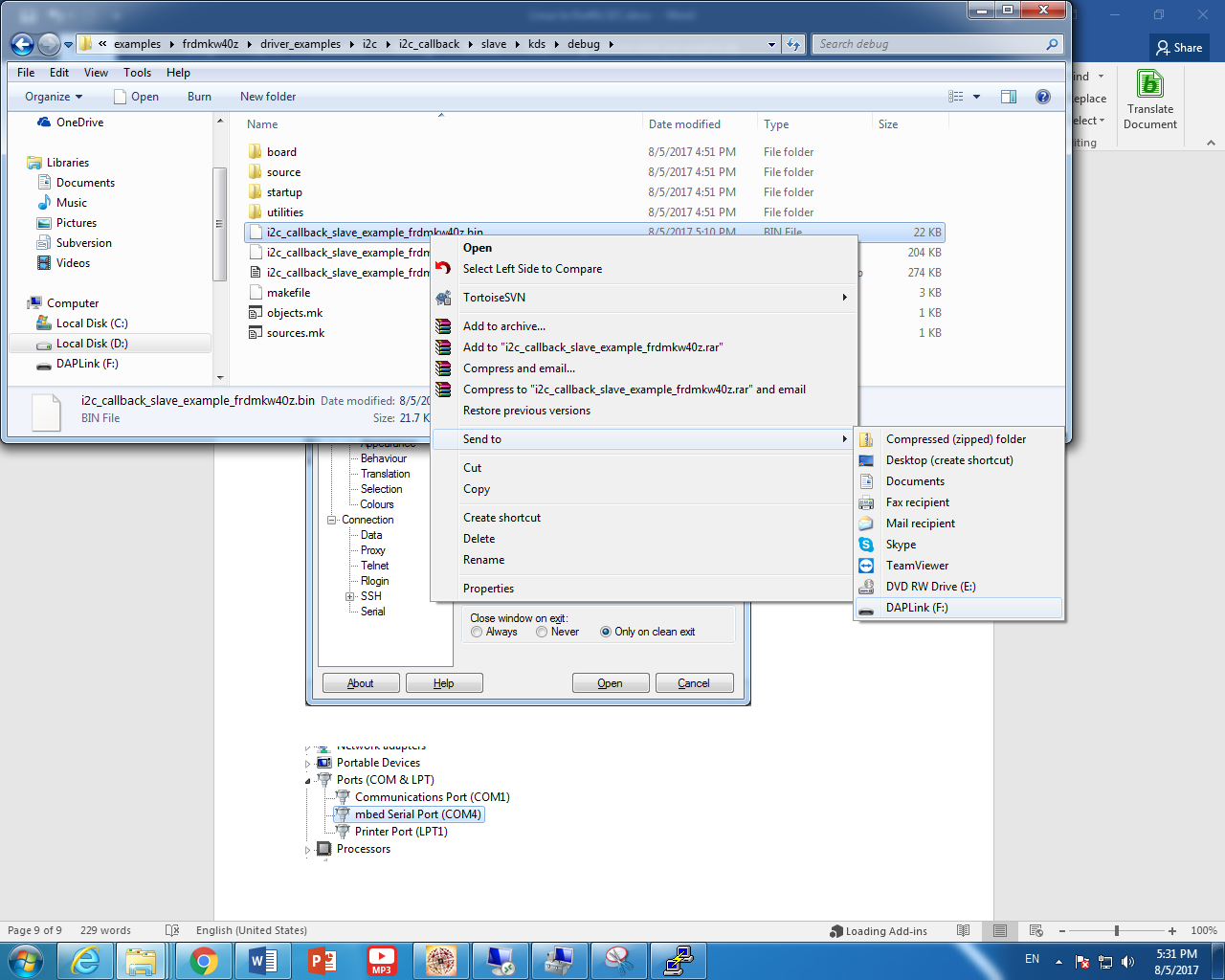
open putty and define the Uart connection:





You can copy and paste the produce binary from the KDS project

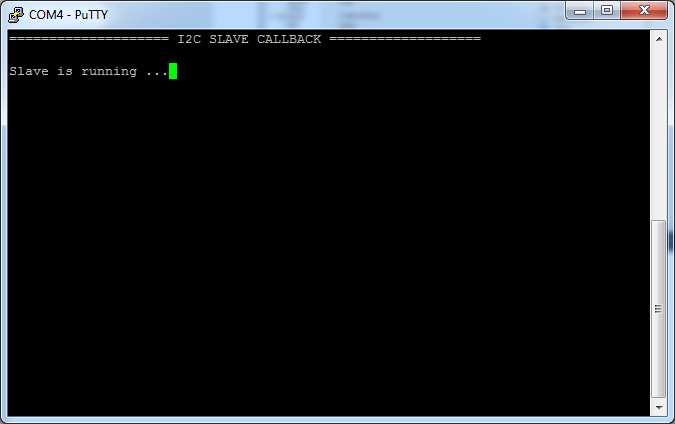
i2c\_callback\_slave\_example\_frdmkw40z.bin using any method you like.



After download the new program, we have two options to reset the board to let it run,

1. Press the reset button
2. Using putty, send a break command

You should see in the putty terminal



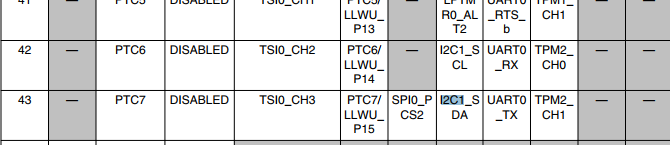
We are now ready to connect the kw40z board to the beaglebone board in the i2c.

Let’s open the datasheet of the 40z

<http://www.nxp.com/docs/en/data-sheet/MKW40Z160.pdf>

I am going to use the I2C1 because I2C0 is already in use for connection to windows in another project.

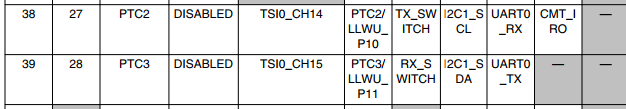
I2C1 can be connected in several places:



PTC6 and PTC7 as alternate 3

PTB16 and PTB17 as alternate 3

And:



As PTC2 and PTC 3 as alternate 3

The Kinetis SDK from nxp become so easy to use that it is amazing

They did an amazing work from the days we use to work with processor expert.

Following the code example, we can see that we already have a function to configure the pin mux

**configure\_i2c\_pins**

We can see that PTC2 which is the i2c1\_scl is already muxed:

PORT\_HAL\_SetMuxMode(PORTC,2u,*kPortMuxAlt3*);

PORT\_HAL\_SetPullCmd(PORTC,2u,true);

PORT\_HAL\_SetPullMode(PORTC,2u,*kPortPullUp*);

PORT\_HAL\_SetPassiveFilterCmd(PORTC,2u,false);

/\* PORTC\_PCR3 \*/

PORT\_HAL\_SetMuxMode(PORTC,3u,*kPortMuxAlt3*);

PORT\_HAL\_SetPullCmd(PORTC,3u,true);

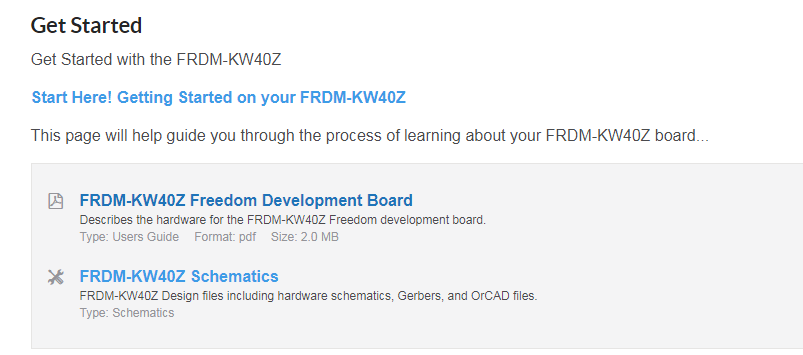
PORT\_HAL\_SetPullMode(PORTC,3u,*kPortPullUp*);

PORT\_HAL\_SetPassiveFilterCmd(PORTC,3u,false);

So it match the data sheet.

Next is to find the i2C on the user connectors

We need the schamtic of the board to search PTC2 and PTC3



<http://www.nxp.com/downloads/en/schematics/FRDM-KW40Z-C2SCH.pdf>

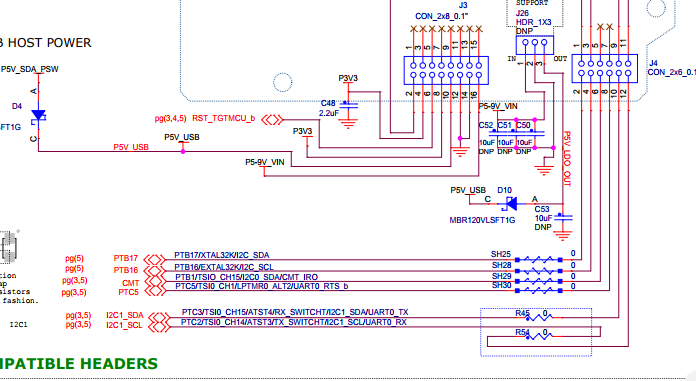
I connected red wire to J4 pin 12 which is I2C1\_SCL

I connected orange wire to J4 pin 10 which is I2C1\_SDA

Also, I need ground.

J3 pin 14 and 16 are connected to ground.

Connected brown to pin 16 at J3.



From the code, we can see that the slave address is 0x7F

.address = 0x7FU,

i2c\_slave\_user\_config\_t userConfig =

{

.address = 0x7FU,

.slaveCallback = i2c\_slave\_callback,

.callbackParam = &callParam,

.slaveListening = true,

**#if** FSL\_FEATURE\_I2C\_HAS\_START\_STOP\_DETECT

.startStopDetect = true,

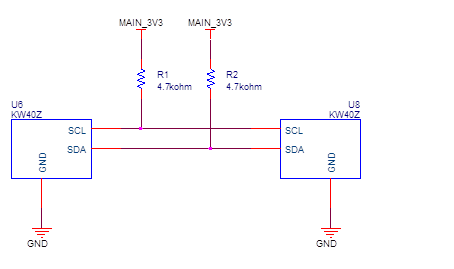
**#endif**

**#if** FSL\_FEATURE\_I2C\_HAS\_STOP\_DETECT

.stopDetect = true,

**#endif**

};



Some notes about the code

I use my own byte fifo to read and write data.

**int** fifo\_in\_read\_index = 0;

**int** fifo\_in\_write\_index = 0;

uint8\_t FIFO\_IN[FIFO\_IN\_SIZE];

**int** fifo\_out\_read\_index = 0;

**int** fifo\_out\_write\_index = 0 ;

uint8\_t FIFO\_OUT[FIFO\_OUT\_SIZE];

some helper functions:

**int** **GetFifoInSize**()

{

**if** (fifo\_in\_read\_index == fifo\_in\_write\_index)

**return** 0;

**if** (fifo\_in\_write\_index > fifo\_in\_read\_index)

**return** fifo\_in\_write\_index - fifo\_in\_read\_index;

**return** FIFO\_IN\_SIZE - fifo\_in\_read\_index + fifo\_in\_write\_index;

}

**int** **GetFifoOutSize**()

{

**if** (fifo\_out\_read\_index == fifo\_out\_write\_index)

**return** 0;

**if** (fifo\_out\_write\_index > fifo\_out\_read\_index)

**return** fifo\_out\_write\_index - fifo\_out\_read\_index;

**return** FIFO\_OUT\_SIZE - fifo\_out\_read\_index + fifo\_out\_write\_index;

}

**int** **IsFifoInFull**()

{

**int** size = GetFifoInSize();

**if** (size < (FIFO\_IN\_SIZE - 1))

**return** 0;

**return** 1;

}

I modified the function

**I2C\_DRV\_SlaveIRQHandler in fsl\_i2c\_slave\_driver.c**

**void** **I2C\_DRV\_SlaveIRQHandler**(uint32\_t instance)

{

assert(instance < I2C\_INSTANCE\_COUNT);

I2C\_Type \* base = g\_i2cBase[instance];

uint8\_t i2cData = 0x00;

bool doTransmit = false;

bool addressed = I2C\_HAL\_GetStatusFlag(base, *kI2CAddressAsSlave*);

bool stopIntEnabled = false;

**#if** FSL\_FEATURE\_I2C\_HAS\_START\_STOP\_DETECT

bool startDetected = I2C\_HAL\_GetStartFlag(base);

bool startIntEnabled = I2C\_HAL\_GetStartStopIntCmd(base);

bool stopDetected = I2C\_HAL\_GetStopFlag(base);

stopIntEnabled = startIntEnabled;

**#endif**

**#if** FSL\_FEATURE\_I2C\_HAS\_STOP\_DETECT

bool stopDetected = I2C\_HAL\_GetStopFlag(base);

stopIntEnabled = I2C\_HAL\_GetStopIntCmd(base);

**#endif**

/\* Get current runtime structure \*/

i2c\_slave\_state\_t \* i2cSlaveState = (i2c\_slave\_state\_t \*)g\_i2cStatePtr[instance];

/\* Get current slave transfer direction \*/

i2c\_direction\_t direction = I2C\_HAL\_GetDirMode(base);

**#if** FSL\_FEATURE\_I2C\_HAS\_START\_STOP\_DETECT

/\*--------------- Handle START ------------------\*/

**if** (startIntEnabled && startDetected)

{

I2C\_HAL\_ClearStartFlag(base);

I2C\_HAL\_ClearInt(base);

**return**;

}

**#endif**

**#if** FSL\_FEATURE\_I2C\_HAS\_START\_STOP\_DETECT || FSL\_FEATURE\_I2C\_HAS\_STOP\_DETECT

/\*--------------- Handle STOP ------------------\*/

**if** (stopIntEnabled && stopDetected)

{

I2C\_HAL\_ClearStopFlag(base);

I2C\_HAL\_ClearInt(base);

**if**(!i2cSlaveState->slaveListening)

{

/\* Disable I2C interrupt in the peripheral.\*/

I2C\_HAL\_SetIntCmd(base, false);

}

**return**;

}

**#endif**

/\* Clear I2C IRQ.\*/

I2C\_HAL\_ClearInt(base);

/\*--------------- Handle Address ------------------\*/

/\* Addressed only happens when receiving address. \*/

**if** (addressed) /\* Slave is addressed. \*/

{

/\* Master read from Slave. Slave transmit.\*/

**if** (I2C\_HAL\_GetStatusFlag(base, *kI2CSlaveTransmit*))

{

/\* Switch to TX mode\*/

I2C\_HAL\_SetDirMode(base, *kI2CSend*);

doTransmit = true;

}

**else** /\* Master write to Slave. Slave receive.\*/

{

/\* Switch to RX mode.\*/

I2C\_HAL\_SetDirMode(base, *kI2CReceive*);

/\* Read dummy character.\*/

I2C\_HAL\_ReadByte(base);

}

}

/\*--------------- Handle Transfer ------------------\*/

**else**

{

/\* Handle transmit \*/

**if** (direction == *kI2CSend*)

{

**if** (I2C\_HAL\_GetStatusFlag(base, *kI2CReceivedNak*))

{

/\* Switch to RX mode.\*/

I2C\_HAL\_SetDirMode(base, *kI2CReceive*);

/\* Read dummy character to release bus \*/

I2C\_HAL\_ReadByte(base);

}

**else** /\* ACK from receiver.\*/

{

doTransmit = true;

}

}

/\* Handle receive \*/

**else**

{

/\* Get byte from data register \*/

i2cData = I2C\_HAL\_ReadByte(base);

**if** (IsFifoInFull() == 0)

{

FIFO\_IN[fifo\_in\_write\_index] = i2cData;

fifo\_in\_write\_index = (fifo\_in\_write\_index + 1) % FIFO\_IN\_SIZE;

} **else** {

PRINTF("fifo in full\n");

}

}

}

/\* DO TRANSMIT\*/

**if** (doTransmit)

{

**if** (GetFifoOutSize() == 0)

{

i2cData = 0x0;

} **else** {

i2cData = FIFO\_OUT[fifo\_out\_read\_index];

fifo\_out\_read\_index = (fifo\_out\_read\_index + 1) % FIFO\_OUT\_SIZE;

}

I2C\_HAL\_WriteByte(base, i2cData);

}

}

I did not used the original code to send data using callback, instead , I store the bytes in fifo in and fifo out.

The idea is that when an interrupt of write to the slave arrive , the byte is being stored in a in\_fifo.

When the interrupts ends, the main loop reads the size of that fifo and pull byte by byte if exists.

It can also do smart mempy.

Above that fifo we can build a protocol for send and receive commands.

In the main function I just echo the write and read back to host:

**while**(1)

{

**if** (((size = GetFifoInSize()) > 0) &&

(GetFifoOutSize() < FIFO\_OUT\_SIZE))

{

**for** (**int** i = 0 ; i < size ; i++)

{

uint8\_t x = FIFO\_IN[fifo\_in\_read\_index];

fifo\_in\_read\_index = (fifo\_in\_read\_index + 1) % FIFO\_IN\_SIZE;

FIFO\_OUT[fifo\_out\_write\_index] = x;

fifo\_out\_write\_index = (fifo\_out\_write\_index + 1) % FIFO\_OUT\_SIZE;

}

}

}

Sources:

In my GitHub

I save the project changes.

You need to take the Kinetis SDK 1.3 for k240z and merge the changes of the i2c\_callback slave example.