**Application Execute in Place in QPSI  
TRENZ TE0722**

# Introduction

This document discusses what was done to get an application to execute in place from QSPI flash on the TRENZ TE0722 board.

We have attempted to keep these notes simple. Much of the detail is in the documents referenced below; here we describe my experiences getting the application into flash and running.

# References.

<http://www.wiki.xilinx.com/Zynq-7000+AP+SoC+Boot+-+Booting+and+Running+Without+External+Memory+Tech+Tip>

<http://www.wiki.xilinx.com/Zynq-7000+AP+SoC+-+AMP+Solution+without+External+Memory+Tech+Tip>

<https://www.xilinx.com/support/documentation/application_notes/xapp1176-xip-axi-quad-spi-ipi.pdf>

# Vivado and SDK version

The work was initially carried out on 2014.4 but we have now migrated to 2016.2. The example SDK project is for the latter.

# ZYNQ Customise Requirements

There is a minimum requirement for the ZYNQ :

* That the UARTs be enabled and connected to an external source so debug messages can be read.
* That the QSPI flash be enabled.

# FSBL Starting Point

We started by creating an FSBL using the build in variation using File->New->Application Project then picking the ‘Zynq FSBL’ from the list of templates.

We then copied the sources from the Trenz FSBL - download at <http://www.trenz-electronic.de/download/d0/Trenz_Electronic/d1/TE0722/d2/Reference_Design.html>.

# The FSBL

## Attempts at Execute in Place

Attempts to get the FSBL to execute in place failed, despite trying on numerous occasions with variations to the approach. We attempted to follow the guidance in the above documents each time.

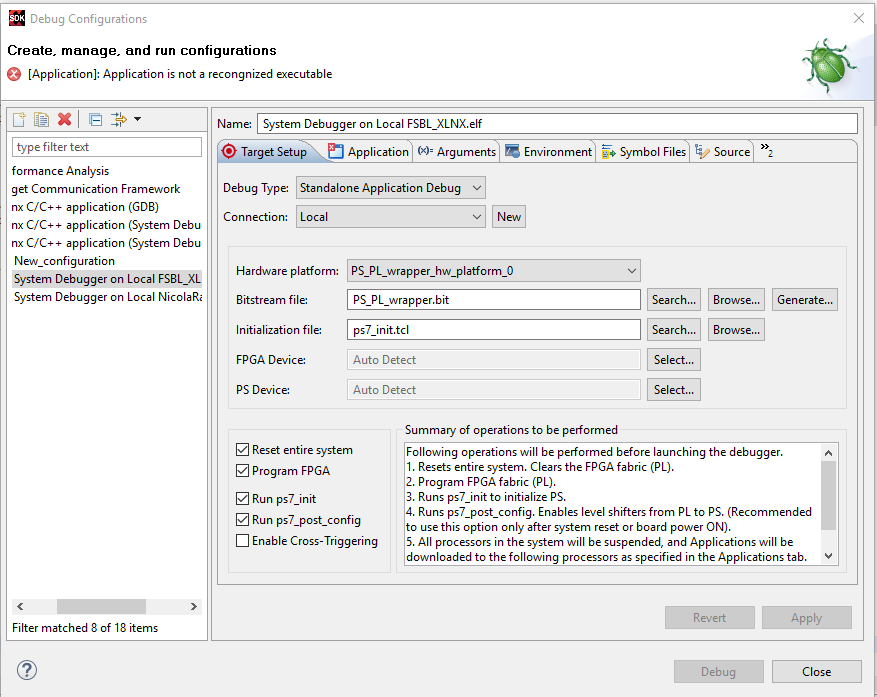
However, if the following discussion is understood then applying the lessons learnt may well allow this to be implemented. Here, having got the application to run from flash, we feel we need to go no further at present.

## Running in SDK ‘Debug Mode’

Running the FSBL in debug mode through the SDK gave a number of issues associated with loading the bit file to the PL and the associated setup.

WE got round this in the following manner in the “Debug Configuration”:

* Make sure ‘Standalone Application Debug’ is selected as the Debug Type
* ‘Reset Entire System’
* ‘Program FPGA’ will download the bit file
* Run PS7\_init.c



Then, in image\_mover.c, the code at about line 304 onwards that downloads the bit file to the PL was bypassed.

**while** (PartitionNum < PartitionCount) {

fsbl\_printf(DEBUG\_INFO, "Partition Number: %lu\r\n", PartitionNum);

// test MultiBootReg reg for jtag

// if testing through a jtag download, make sure teh bit image is loaded

// after a reset and remove the comment within the braces.

// MAKE SURE YOU PUT THE COMMENTS BACK BEFORE A BOOTGEN.

**if** ( PartitionNum == 1 ) // if running debug mode from the SDK - remove otherwise

{

PartitionNum++; // remove comment only if debugging through JTAG

**continue**; // ditto

}

HeaderPtr = &PartitionHeader[PartitionNum];

…

Note that the code in image\_mover.c about line 221 that determines the location in flash of the ‘ImageStartAddess’ variable gave some problems. For some time the offset into flash was calculated to 0x200000 and clearly caused the FSBL to crash when flash was accessed. I forced this offset to 0 and was able to continue work. In later work, this problem seems to have ‘gone away’; but beware. We have left the code there but commented out.

## Loading the FSBL from Flash

This presented no real problems – but be aware that no diagnostic messages will be sent to the UART until the bit file is loaded to the PL and initialised. This gave me some problems for a while hence the need above to run the FSBL as a download project.

## Using L2 Cache

This was discontinued early in the process in order to simply get the application to run. I am sure this could now be done by following the description in the original documents.

# The Application

The big stumbling block during was the layout of the ELF image and the resulting .Bit file and getting the components to the right place.

Further problems were then encountered with the alignment of some components within the bitsteam.

## The Linking Phase

This proved to be the big stumbling block and took a long time and a lot of effort to get something to work in the required manner.

If the application is to run from the ASPI flash then there are 3 initial requirements:

* That the application code be linked to run from a specific address (in flash)
* That the application code be burned to flash at that location
* That the application data, heap and stack to allocated to ram – in our case the on-chip ram

Provided the QSPI is generated then a flash region is included in the MEMORY sections.

This can be modified to place the FLASH region at a location where the code will be placed:

/\* ps7\_qspi\_linear\_0\_S\_AXI\_BASEADDR : ORIGIN = 0xFC000000, LENGTH = 0x1000000 \*/

FLASH : ORIGIN = 0xFC400000, LENGTH = 0xC00000

So in this case the application code is linked to run from 0xFC400000.

Following the example, the first attempts involved introducing the link header for the .text region:

PHDRS {

text PT\_LOAD;

}

.text : {

\*(.vectors)

\*(.boot)

…

\*(.ARM.extab)

\*(.gnu.linkonce.armextab.\*)

} > FLASH : text

Together with assigning the various data regions as per the example link file. This was OK BUT the headers with the data section were located before the text section in the ELF file, which in turn resulted in the code being in the wrong place in the .bit file and hence in flash – clearly this failed.

We got round this by adding a further header section as follows:

PHDRS {

texthdr PT\_LOAD;

datahdr PT\_LOAD;

}

Then any data sections that were not placed in flash by the AT> FLASH suffix were placed through this header. See the linker file in the ‘NicolaRadio’ build in the code.

## Building the Bitstream

Once we had established the link headers were putting the application code and data into the appropriate place, then we were able to build the bit stream.

Note that when you generate a bit stream through the (application)->Create Boot Image, then the filenames generated in the bootgen file are absolute not relative. Hence the .bif file cannot be copied and re-used without modification – I fell down this hole for a few days until I realised the downloaded code was not matching that in the build.

So, if we are running the application code from 0x400000 offet in flash, then the bootgen file looks as follows:

//arch = zynq; split = false; format = BIN

the\_ROM\_image:

{

[bootloader]C:\Users\Pete\Documents\Nicola-Zynq\2016\TE0722TD\TE0722TD.sdk\FSBL\_XLNX\Debug\FSBL\_XLNX.elf

C:\Users\Pete\Documents\Nicola-Zynq\2016\TE0722TD\TE0722TD.sdk\PS\_PL\_wrapper\_hw\_platform\_0\PS\_PL\_wrapper.bit

[offset = 0x400000]C:\Users\Pete\Documents\Nicola-Zynq\2016\TE0722TD\TE0722TD.sdk\NicolaRadio\Debug\NicolaRadio.elf

}

Noting that we are not running the FSBL as XIP and that the offset is specified for the application code.

The flash can then be burnt and executed.

# Running the Application

So, having got the elf file to link correctly and then built and burned to flash the .bin file, why did not the system work?

Here is where running the FSBL from the SDK and being able to single step the code during the hand-over from the FSBL to the application became very important. I set a breakpoint at the handover call in the FSBL and single stepped through the code.

The problem we found was that the memory mapping was not loading correctly and the ARM chip crashed as soon as the instruction was executed.

This was traced to the memory map not be correctly aligned; somewhere along the line the alignment of the mmu\_tbl had been lost. So correcting this in the linker script:

.mmu\_tbl (ALIGN(0x4000)): {

\_\_mmu\_tbl\_start = .;

\*(.mmu\_tbl)

\_\_mmu\_tbl\_end = .;

} > FLASH : texthdr

Got us up and running and hallelujah the “HELLO WORLD” message appeared.

## Getting the FREERtos to Run

The final step was to get our real time OS of choice to run.

But as soon as we tried to initialise the RTOS it objected to its database not being configured.

The code to copy the data block from the flash to the ram had been included:

copy(&\_dataLMA\_A,&\_dataVMA\_start\_A,&\_dataVMA\_end\_A);

but was clearly not working properly.

After much searching around and looking at memory dumps, we determined that the master data section in flash was aligned and that we needed to skip into the correct address. This has been implemented as follows:

**unsigned** **int** \*modifiedLMAAddress;

modifiedLMAAddress = (**unsigned** **int** \*) &\_dataLMA ;

**while** ( \*modifiedLMAAddress == 0xFFFFFFFF ) modifiedLMAAddress++;

copy( (**char** \*) modifiedLMAAddress, &\_dataVMA\_start, &\_dataVMA\_end);

And works. The example code now starts the RTOS, kicks off three tasks and appears to work successfully.

(In writing this document it may be a better solution in the long run will be to align the data section in the linker as per the mmu\_tbl above. We will try this later on.)

# Conclusion

We now have a system we can build, burn to flash and execute. The above may not be elegant and indeed might not be a full correct solution, but we now can proceed with our development work.

If this document is of any help or you need any further comments please contact me at [peteallwright@btinternet.com](mailto:peteallwright@btinternet.com)

Pete Allwright

31 October 2016.