

SPU TC2 Firmware Guide

Last revised 7/21/23

SPU Overview : Introduction

- The SPU is a self-contained sparse neural network accelerator with 1 MB on-board SRAM to store parameters
 - While the chip is powered down, parameters are to be kept in off-SPU nonvolatile storage
- The SPU connects to a host processor over SPI
 - The host manages the state of the SPU, loads network parameters, and exchanges neural network inputs and outputs with the SPU
- While the network is running, parameters are not transferred, only input and output data.
 - Multiple networks can be run on the SPU simultaneously—they just all have to fit.

SPU Overview : Major Components

- **SPI Controller**

- Works using only the SPI clock itself, functional even if the PLL is off
- Controls system registers, which configure the PLL, the pads, and the SPI controller itself
- Programs, configures, and drives inputs to the SPU cores (this requires the SPU to be clocked through the PLL)

- **Oscillator pad**

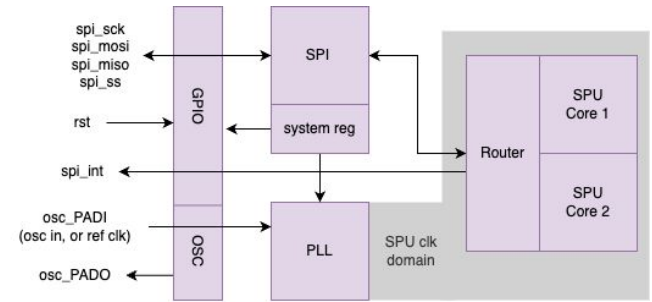
- Can be used to produce a clock from a 32KHz crystal, or can simply be used as an input pad for a reference clock

- **PLL**

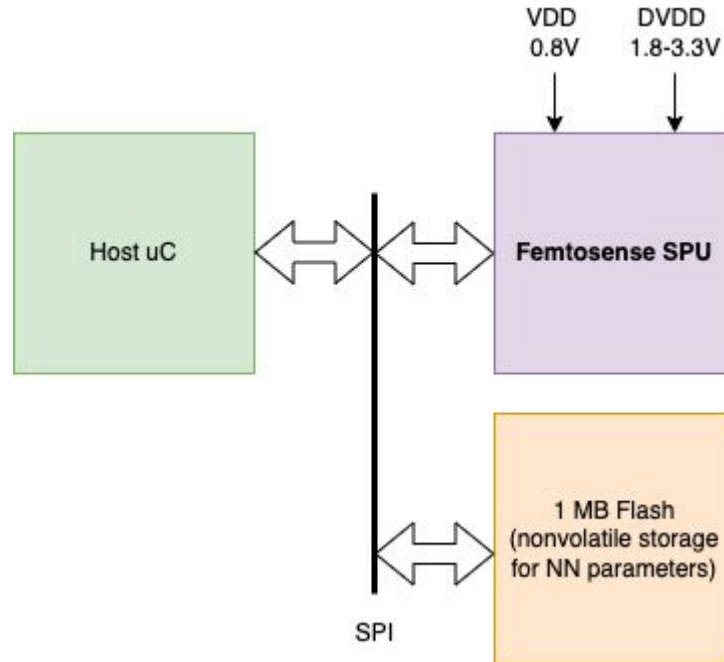
- Multiplies the clock coming from the oscillator pad, max multiplier 8192, bypassable

- **SPU**

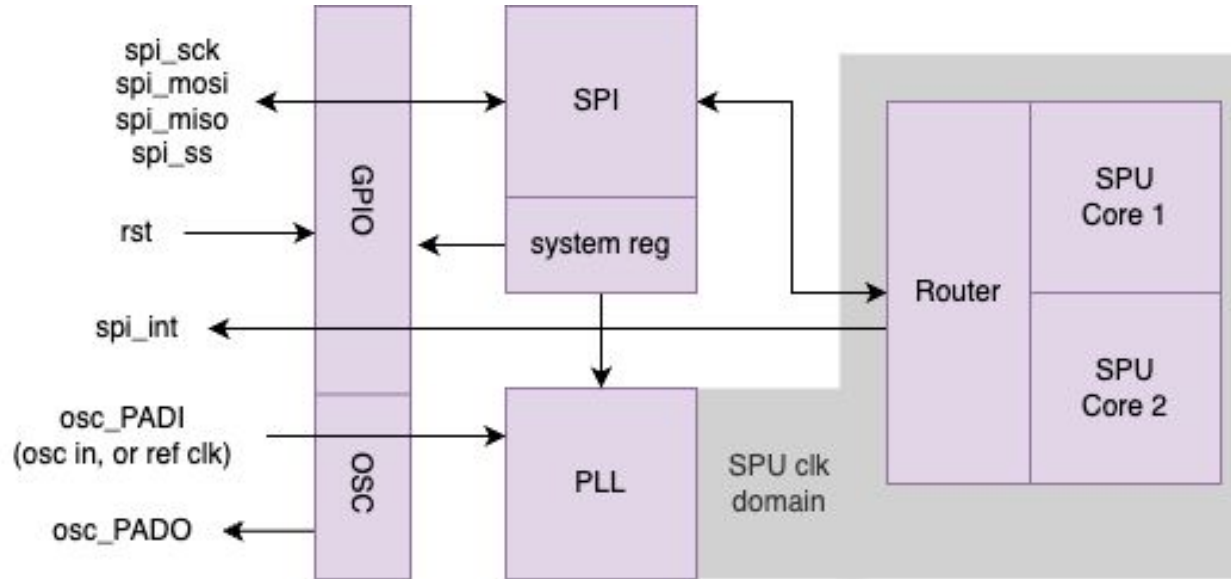
- Contains two interconnected cores, each with 4 SIMD datapaths and 0.5MB of data memory
- Clocked by the PLL output, only accessible when a clock is present
- Memories can be put into retention or shut down to save power



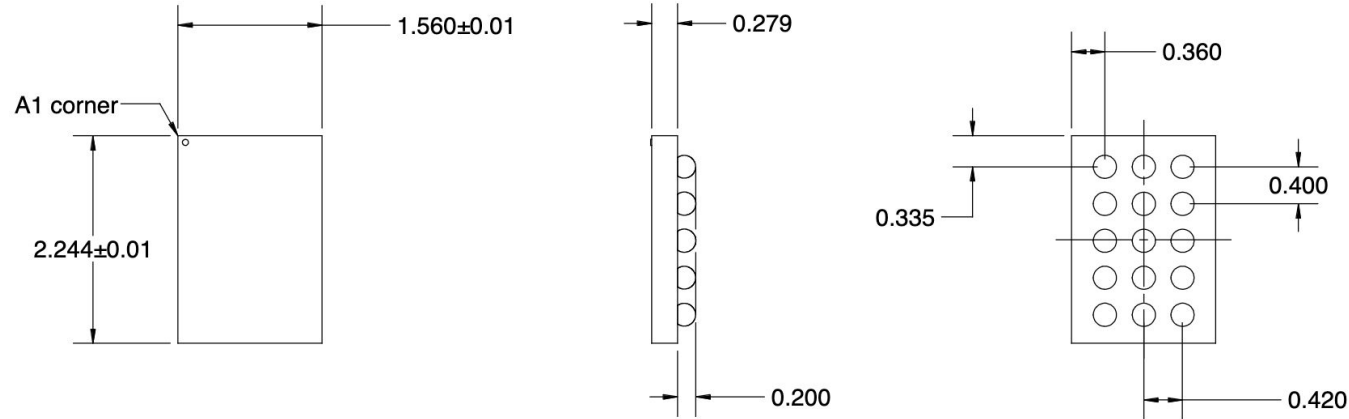
SPU Overview : Typical System Diagram



SPU Overview : Internal Organization



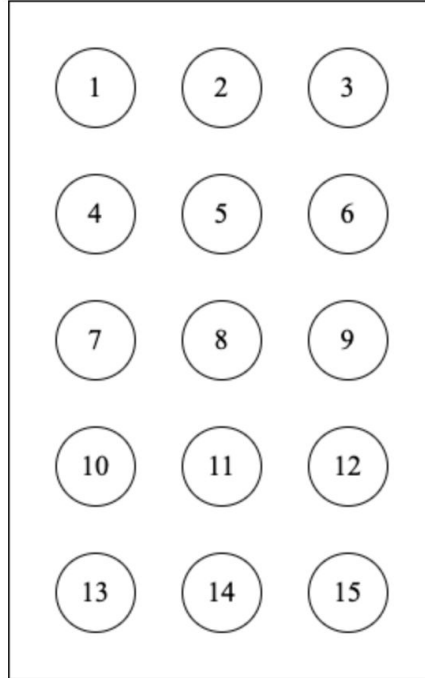
SPU Overview : POD



All dimensions in the drawing are mm. View is from **bottom** (ball array-side) of chip. Ball array is a regular grid at 420um x-pitch, 400um y-pitch. Note offset to pin 1: array is centered in width of the chip but slightly off-center in its height. Ball size is 250um. Recommended PCB pad size is 227um. Die thickness (not including balls) is 11 mil.

SPU Overview : Pinout

PIN #	ID
1	SPI_MOSI
2	VDD
3	VSS
4	SPI_SCK
5	SPI_MISO
6	RST
7	SPI_SS
8	DVDD
9	VSSA
10	SPI_INT
11	VDDA
12	OSC_PADI
13	VSS
14	VDD
15	OSC_PAD

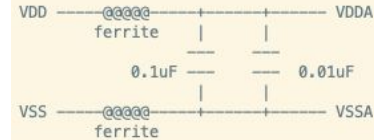


Note: view is looking “down” through top of chip—reflects PCB footprint

SPU Overview : Power Supplies

- DVDD : 1.8 - 3.3V IO Power
- VDD : 0.8V nominal core power
- VDDA : 0.8V PLL power
 - The PLL is ideally supplied by an isolated power supply. We have been advised that this isolation is not usually necessary for low power parts. EVK2 omits the ferrites completely but preserves the .1uF and .01uF caps. The VSS ferrite is less important than the VDD one. See ideal isolation circuit to the right:

The PLL's two analog supplies should be filtered with two series ferrite beads and two shunt 0.1uF and 0.01uF capacitors. The ferrite on VSS is preferred but optional. Adding the ferrite on VSS converts supply noise to substrate noise as seen by the PLL. The PLLs are designed to be relatively insensitive to supply and substrate noise, so the presence of this ferrite is a second order issue.



The ferrite beads should be similar one of the following from Murata:

Part number	R@DC	Z@10MHz	Z@100MHz	Z@1GHz	size
BLM18EG601SN1 *	0.35	200	600		0603
BLM18PG471SN1	0.20	130	470		0603
BLM18KG601SN1	0.15	160	600		0603
BLM18AG601SN1	0.38	180	600		0603
BLM18AG102SN1	0.50	280	1000		0603
BLM18TG601TN1	0.45	190	600		0603
BLM15AG601SN1	0.60	200	600		0402
BLM15AX601SN1 *	0.34	190	600		0402
BLM15AX102SN1	0.49	250	1000		0402
BLM03AX601SN1	0.85	120	600		0201

* preferred choice

Similar ferrite beads are also available from Panasonic. The key characteristics to select are:

- DC resistance less than 0.40 ohms
- impedance at 10MHz equal to or greater 180 ohms
- impedance at 100MHz equal to or greater than 600 ohms

The capacitors should be mounted as close to the package balls as possible.

Using the SPU

SPU Usage : Basic Usage Outline

Part 1: Compile the model and load to NVM

1. Compile the neural network model, or use a Femosense-provided precompiled model. See femtohavpub and femtocrux guides. Outputs:
 - Compiled model bitstreams, to be handled by firmware (initial release: SD programming files)
 - Mailbox identifiers for inputs and outputs (see terminal output from femtohavpub). This is always the same for single models with one input and output.
2. Load the model binary in the NVM (initial release: the SD card. Future: the flash chip on the EVK board)

SPU Usage : Basic Usage Outline

Part 2: Run the SPU

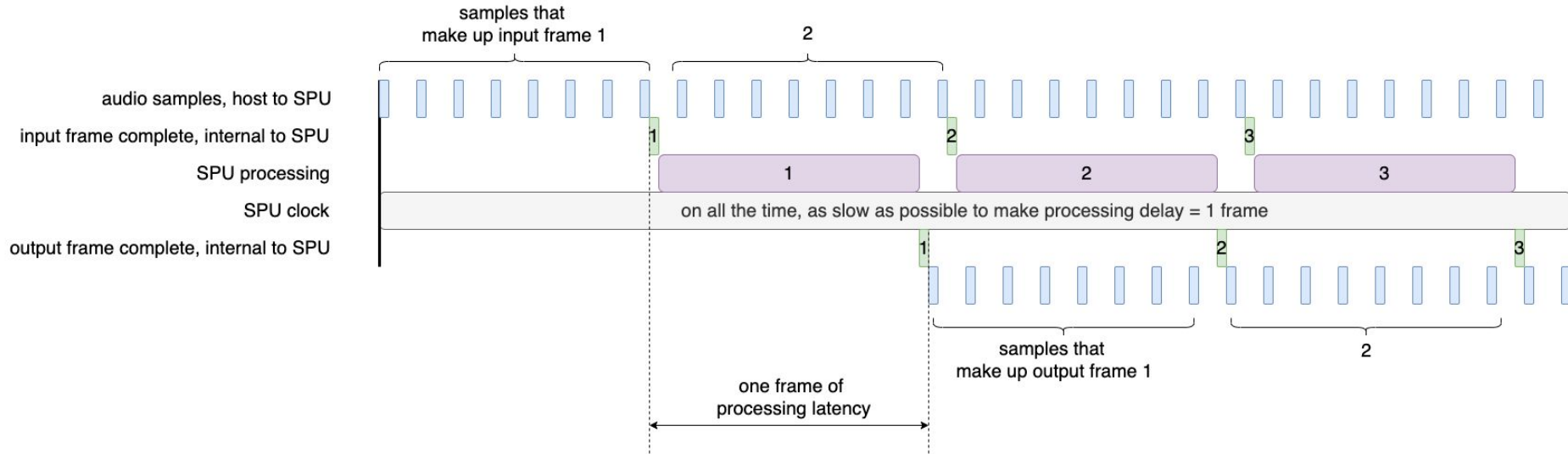
1. Power on the SPU
2. Configure the osc pad + PLL to produce the application's required frequency (per information from the compiler and desired duty cycle. See subsequent pages and reference firmware)
3. Transfer compiled bitstream (contains neural network parameters + other config) from NVM to the SPU
4. Streaming data. Repeat:
 - a. Send input frame (e.g. audio spectral frame, or chunk of time-series data) to the appropriate mailbox. The SPU will automatically start processing the frame
 - b. Wait for the SPU to finish processing. Either wait for the SPI_INT interrupt or sleep for a fixed amount of time.
 - c. Receive output frame

SPU Usage : Duty Cycle Considerations

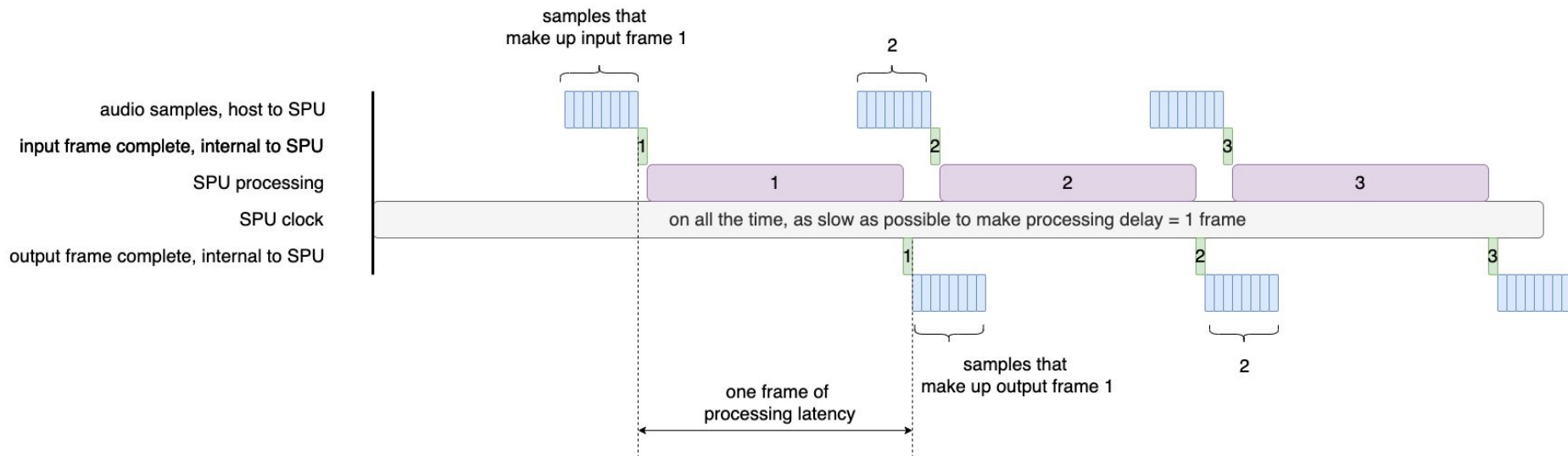
- More variations are possible, but the following slides present a few possible operating modes
- In “**Steady Processing, Steady IO**”, the SPU is processing almost all the time. The host is regularly sending/receiving audio samples to/from the SPU (e.g. as soon as they are sampled/just before they are needed). The SPU buffers the received audio data into frames (one set of neural network inputs) and starts processing when one is complete, producing a complete frame that is drained as outputs are sent. This style of processing requires the lowest max frequency for the SPU possible, limiting peak current. But it incurs the maximum processing latency of a single frame’s duration.
- In “**Steady Processing, Bursting IO**”, the SPU is still processing all the time, but the IO events are more clustered. This requires the host to buffer up data, but might be more convenient.
- In “**Bursting processing, Bursting IO**”, the SPU clock is boosted during the processing phase to try to reduce the processing latency as much as possible.

CAVEAT: (Steady, Bursting) is effectively what the EVK examples ship with. (Bursting, Bursting) is possible with firmware manipulation. (Steady, Steady) requires a compiler update to be shipped later.

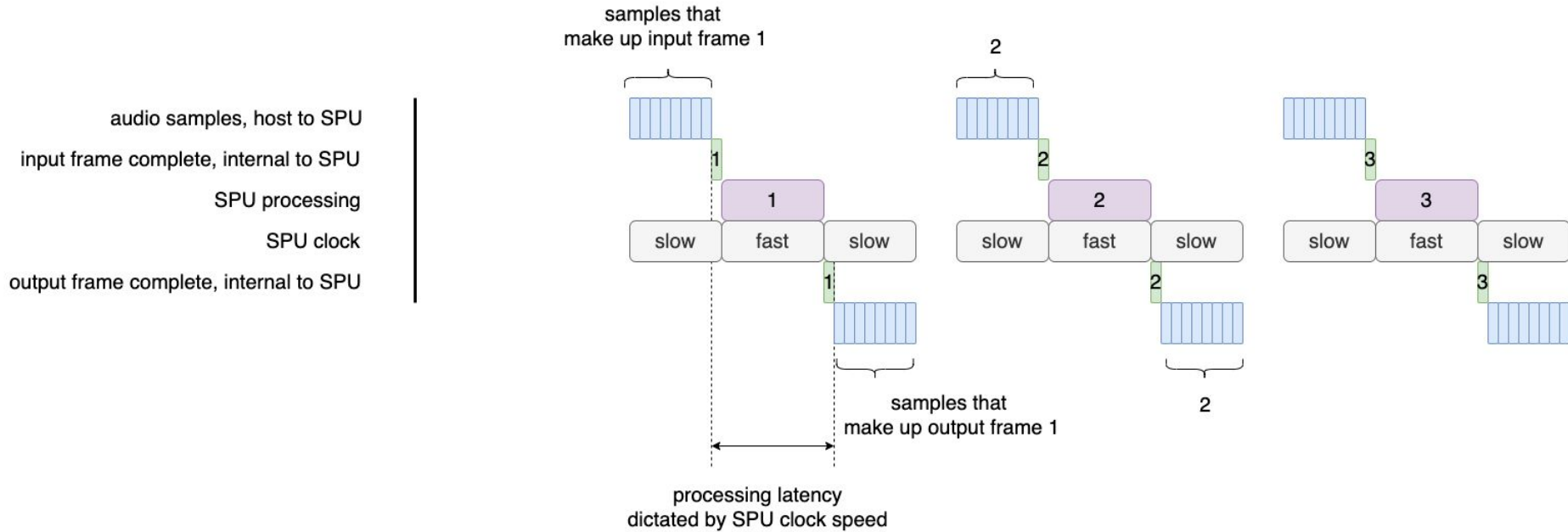
"Steady Processing, Steady IO"



"Steady Processing, Bursting IO"



"Bursting Processing, Bursting IO"



SPU Usage : Clocking Considerations

- The PLL takes a maximum of 500 reference clock cycles to lock to a new frequency
 - Practical re-lock time TBD
 - This is only applicable if the VCO frequency is modified. Modifying the output dividers or gating the clock is practically instantaneous
 - The VCO frequency should interpolate smoothly if the frequency is modified, but exact frequency is undefined until lock

Clocking Schemes

- Reference clock, no multiplication (max core frequency ~ 80MHz, pad limited)
 - Oscillator off, bypassed
 - PLL off, bypassed
- Reference clock, multiplication (max core frequency ~ 500MHz, timing limited)
 - Oscillator off, bypassed
 - PLL on, multipliers set as needed
- Clock from crystal, multiplication (max core frequency = $32\text{KHz} * 8192 = 256\text{MHz}$)
 - Oscillator on
 - PLL on, multipliers set as needed

Femtosecense-Provided Driver APIs

```

//Hop size (number of samples) of SPU input
#define HOP_SIZE 256

//Initializes the SPU
int initializeSPU();

//Resets the SPU using the reset pin
void SPUReset(int resetPin);

//Sets up the SPU PLL registers
void SPUPLLSetup();

//Turns on all of the SPU memory banks
void SPUMemorySetup();

/* Request to read SPU data.
 * instruction: SPU instruction
 * address: start address of data requested
 * length: length of data requested
 * data: buffer for returned data from SPU
 */
void SPU_Read(byte instruction, uint32_t address, int length, uint32_t* data);

/* Write data to SPU.
 * instruction: SPU instruction
 * address: start address of write location
 * length: length of data to write
 * data: buffer of data to write to SPU
 */
void SPU_Write(byte instruction, uint32_t address, int length, uint32_t* data);

//Tests basic SPU register reads
int SPUIntegrityCheck();

```

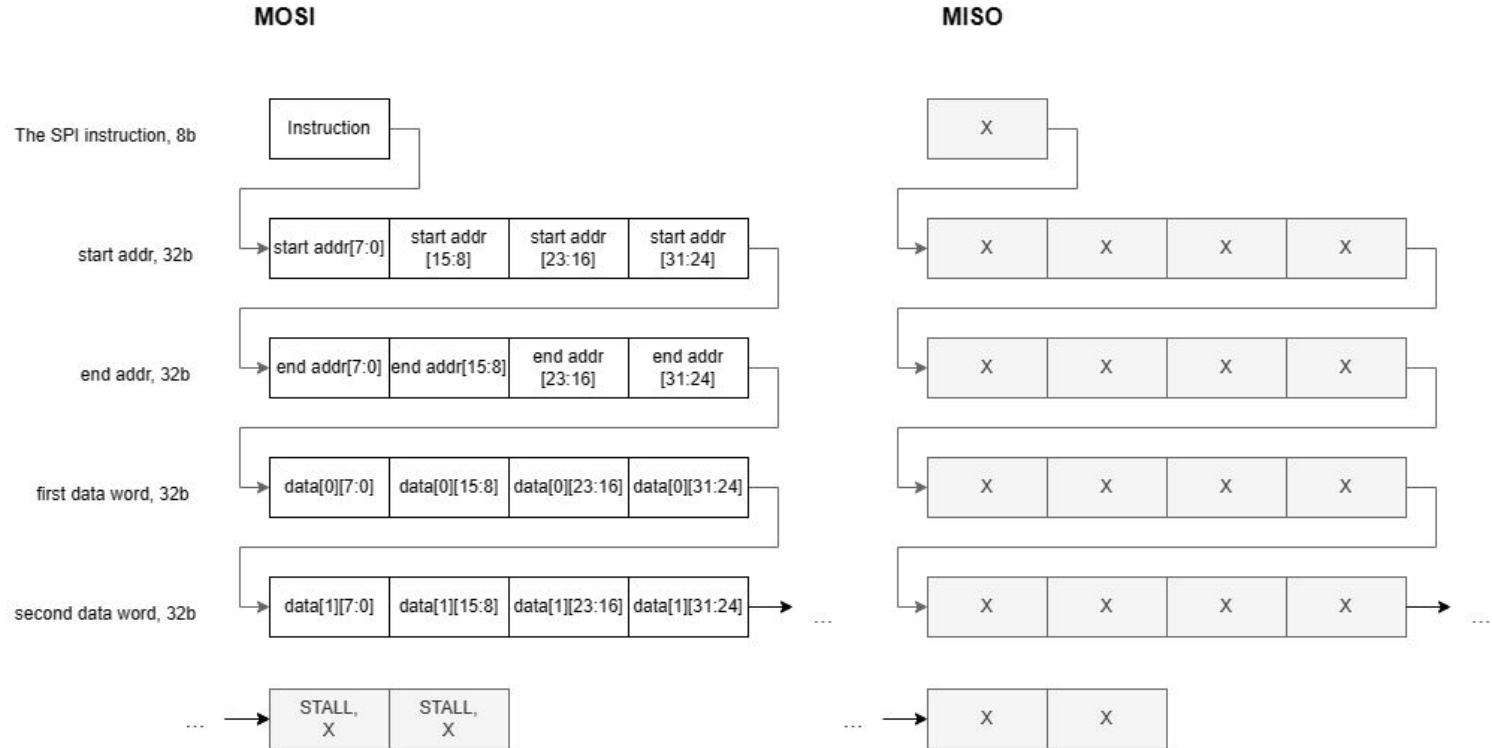
NOTE: the details of nearly all of the following are buried under the provided driver APIs. Understanding the following should not be strictly necessary to use the SPU.

SPI Message Format

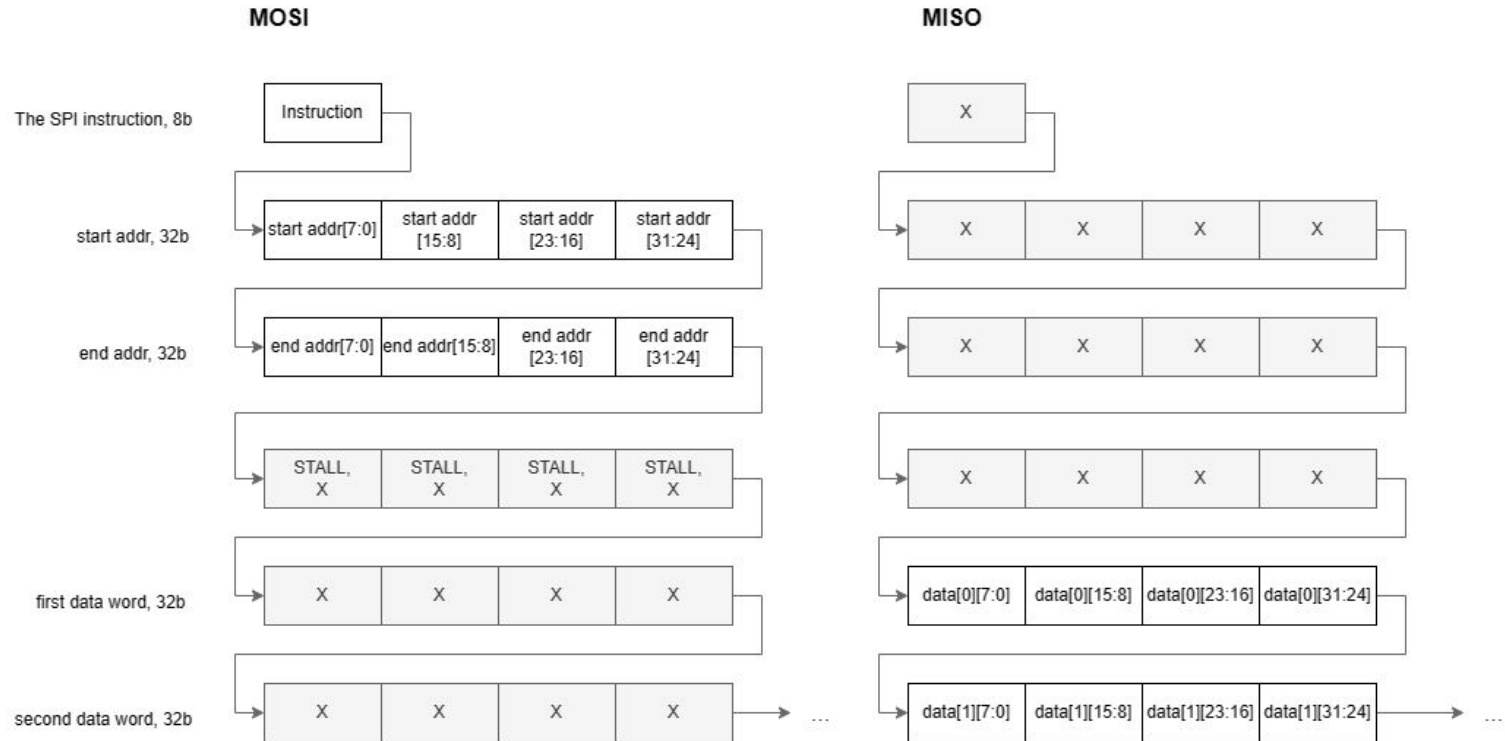
Internally, the SPU has an APB interface. Like the related, more feature-heavy AXI, APB presents a simple address-mapped interface to the host uC.

The SPU wraps this with an SPI interface, which presents a burst-like interface (start addr, end addr, followed by streaming data). This is the external interface available to the user over which all programming and configuration takes place

SPI Write Sequence



SPI Read Sequence



Inst Types

In terms of message sequence, reads/writes to all locations look the same. Internally, the timing is a little different depending on where we're writing to, so we have to tell the SPI controller first thing, in the instruction word.

The SPI module has its own set of registers. In order to access these, SPI_IOREG must be set. The SPI register domain is used to control the pads and PLL.

The chip is programmed using SPI_APB. This is what you use to program memories, control registers, etc. The PLL must be active to talk to these registers.

SPI_AXIS is used to stream data into the router. This is how streaming vectors (input/output data) are sent. These transactions are streams internally. The transaction will stay "open" if SPI_AXIS (not _LAST) is used. Typically, we'll use SPI_AXIS_LAST, assuming the data is sent in one complete transaction. For SPI_AXIS, the absolute values of the start/end aren't important, but their difference needs to match the amount of data. It's easiest to think of it as an index range, 0-N.

Instruction Word Format

MSB	LSB	
x	Inst Type 2b	wr_en 1b

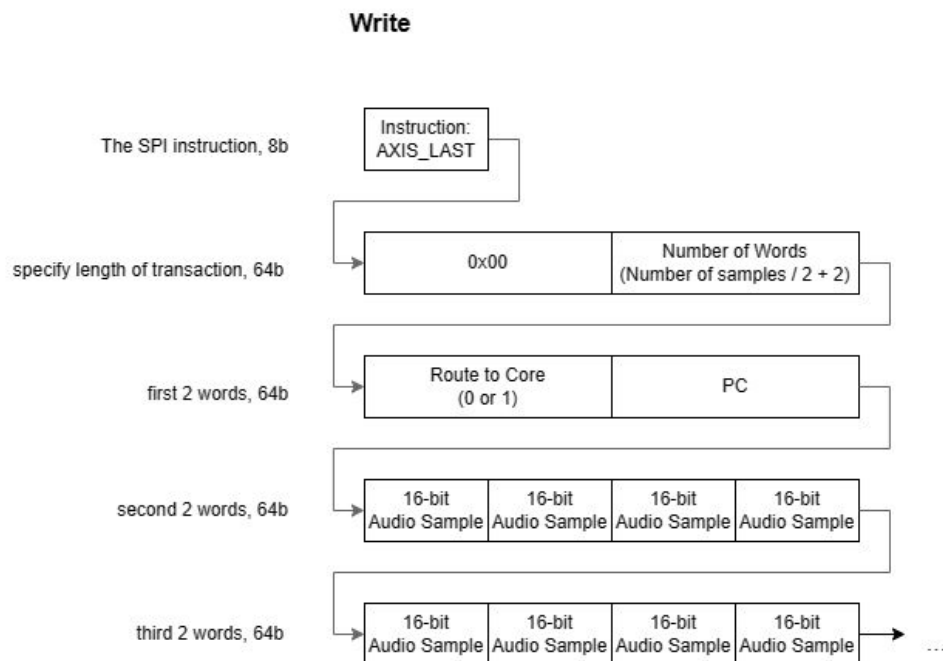
Inst Types

```
typedef enum logic [1:0] {  
    SPI_IOREG=0,  
    SPI_APB=1,  
    SPI_AXIS=2,  
    SPI_AXIS_LAST=3  
} SPIInstType;
```


Router Data Format

When sending data in through AXIS commands, the first 64-bit word contains the route and "next PC". The route is just the core ID that's targeted and the "next PC" is the program counter value of the thread that's supposed to start. These values will be program-dependent. The first 32-bits is the route, and the next 32-bits is the next PC. Subsequent words will be the vector that's sent in (length is program-dependent). This is illustrated below:

AXIS Example



the APB address space is organized in "core-size" chunks: $2^{16} * 8$

The SPU uses the first chunk for conf registers that affect both cores. Note, this is not the same as the SPI conf reg.

The two chunks above that are the primary memory for each core.

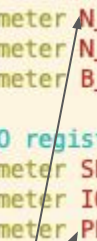
Relative address map for a single core to the right:

```
memory: DM, 64 bits wide
bank: 0
0x00000000 (0) -- 0x0000fff8 (65528)
bank: 1
0x00010000 (65536) -- 0x0001fff8 (131064)
bank: 2
0x00020000 (131072) -- 0x0002fff8 (196600)
bank: 3
0x00030000 (196608) -- 0x0003fff8 (262136)
bank: 4
0x00040000 (262144) -- 0x0004fff8 (327672)
bank: 5
0x00050000 (327680) -- 0x0005fff8 (393208)
bank: 6
0x00060000 (393216) -- 0x0006fff8 (458744)
bank: 7
0x00070000 (458752) -- 0x0007fff8 (524280)
memory: TM, 16 bits wide
bank: 0
0x00080000 (524288) -- 0x00083fff8 (540664)
bank: 1
0x00090000 (589824) -- 0x00093fff8 (606200)
bank: 2
0x000a0000 (655360) -- 0x000a3fff8 (671736)
bank: 3
0x000b0000 (720896) -- 0x000b3fff8 (737272)
memory: SB, 11 bits wide
bank: 0
0x000c0000 (786432) -- 0x000c01f8 (786936)
memory: RQ, 6 bits wide
bank: 0
0x000d0000 (851968) -- 0x000d01f8 (852472)
memory: PB, 10 bits wide
bank: 0
0x000e0000 (917504) -- 0x000e01f8 (918008)
memory: IM, 63 bits wide
bank: 0
0x000f0000 (983040) -- 0x000f1fff8 (991224)
```

SPI Register Space

Register Indexing

```
15 parameter N_PLL_CONF = 5;  
16 parameter N_IO_CONF = 2 + N_PLL_CONF; // number of IO conf regs (SPI clock domain)  
17 parameter B_IO_CONF = 16; // bits per IO conf reg, max BAPB  
18  
19 // IO register array mapping  
20 parameter SPI_CONF_IDX = 0;  
21 parameter IO_CONF_IDX = 1;  
22 parameter PLL_CONF_IDX = 2;
```



Note that PLL conf reg
occupies 5 addresses
There are 7 total reg
locations

Currently, only the Zynq firmware
accesses this space, the python
driver doesn't "know" about it

SPI Addr	Data
0 = SPI_CONF_IDX	SPI Conf Word
1 = IO_CONF_IDX	IO Conf Word (pad conf)
2 = PLL_CONF_IDX	PLL Conf Word 0 Misc PLL control bits
3	PLL Conf Word 1 BWADJ
4	PLL Conf Word 2 CLKOD (output divider factor)
5	PLL Conf Word 3 CLKF (mult factor)
6	PLL Conf Word 4 CLKR (input divider factor)
7	PLL lock (1b, lsb is lock)

SPIConf Word

```
typedef struct packed {  
    logic [B_IO_CONF - 1 - 1 : 0] unused;  
    logic [1 : 0] disable_cores;  
    logic no_half_cycle_delay;  
} SPIConf;
```

Conf reg for the SPI controller itself
Shouldn't need to touch
Can put the SPI in SPI mode 1 (instead of 0)

```
function SPIConf get_SPIConf_default();  
    SPIConf x;  
    x.unused = '0;  
    x.disable_cores = '0;  
    x.no_half_cycle_delay = 1'b0; // default is to have a half-cycle delay, SPI mode 0  
e    return x;
```

IOConf word

MSB

```
typedef struct packed {  
    logic [B_IO_CONF - 11 - 1 : 0] unused;  
    logic      osc_en;  
    logic      osc_test_en;  
    logic      fast_slew_en;  
    logic [1 : 0] drive_strength;  
    logic      pull_en;  
    logic      pull_direction;  
    logic      pad_retention_on;  
    logic      osc_SF1;  
    logic      osc_SF0;  
    logic      ret_on;  
} IOConf;
```

LSB

{0, 1} (default) = Use external clock from FPGA. {1, 0} = make clock using crystal

SF0/SF1 unused for TC2,
Fixed 32KHz osc

FEMTOSENSE

Reset values for pad circuit controls and oscillator controls

consider SYNTHESIS=True for osc_en/osc_test_en defaults

We actually go to the non-SYNTH settings (external clock) on boot in Zynq code right now

```
function IOConf get_IOConf_default();  
    IOConf x;  
    x.unused      = '0;    // don't care  
    // for synth, boot into running mode (still forceable)  
    // for RTL test, go into test mode, skip 30M ns  
    x.osc_en      = 1'b0;  // OSC input, osc off, ref clock active  
    x.osc_test_en = 1'b1;  // OSC input, using external clock  
    x.fast_slew_en = 1'b0;  // pad slew normal  
    x.drive_strength = 2'b01; // second-lowest pad drive  
    x.pull_en     = 1'b0;  // weak pull not enabled  
    x.pull_direction = 1'b0; // don't care  
    x.pad_retention_on = 1'b1; // PVSENSE input, pads will hold last value when not  
    x.osc_SF1      = 1'b0;  // crystal freq range  
    x.osc_SF0      = 1'b0;  // crystal freq range  
    x.ret_on       = 1'b0;  // pad value retention disabled by default  
    return x;  
endfunction
```

Fe

PLLConf Word

Defaults for PLL, 5 words,
See TCI manual, in “Chip Internal”/TCI_guides

```
typedef struct packed {  
    Word 4 logic [B_IO_CONF - 6 - 1 : 0] unused4;  
          logic [5 : 0] CLKR;  
  
    Word 3 logic [B_IO_CONF - 13 - 1 : 0] unused3;  
          logic [12 : 0] CLKF;  
  
    Word 2 logic [B_IO_CONF - 4 - 1 : 0] unused2;  
          logic [3 : 0] CLKOD;  
  
    Word 1 logic [B_IO_CONF - 12 - 1 : 0] unused1;  
          logic [11 : 0] BWADJ;  
  
    Word 0 logic [B_IO_CONF - 5 - 1 : 0] unused0;  
          logic PWRDN;  
          logic BYPASS;  
          logic TEST;  
          logic FASTEN;  
          logic ENSAT;  
          logic RESET;  
} PLLConf;
```

Default multiplier is 500:

$$\begin{aligned} F_mult &= (f + 1) / ((r + 1) * (od + 1)) \\ &= 0x3e8 / [(0x1)(0x2)] \\ &= 1000/2 \\ &= 500 \end{aligned}$$

BWADJ+1 should be half of CLKF+1
 $BWADJ = (CLKF+1)/2 - 1$

```
x.CLKR[5:0] = 'h00;  
x.CLKF[12:0] = 'h03E7;  
x.CLKOD[3:0] = 'h1;  
x.BWADJ[11:0] = 'h1F3;  
  
{x.PWRDN, x.BYPASS, x.TEST, x.FASTEN, x.ENSAT, x.RESET} = '0;
```

IMPORTANT NOTE:

There is a bug in the packing of these 5 words.
When the RESET was added in 1p2, the size of the “unused0”
field was not decremented.

As a result, the 4 upper words are all shifted by a bit.
To program a value of “x”, you should actually set “x << 1”

Memory Power Configuration

To power on memories:

First, put the memory in “CD” state. Write:
0x05098

Then turn each memory on, writing
0x11098

To each of the addresses to the right:

Memory control SPI_APB addresses (8x DM, 4x TM, 1x IM, respectively)

```
00000034 : core 0 : DM_CONF0
00000038 : core 0 : DM_CONF1
0000003C : core 0 : DM_CONF2
00000040 : core 0 : DM_CONF3
00000044 : core 0 : DM_CONF4
00000048 : core 0 : DM_CONF5
0000004C : core 0 : DM_CONF6
00000050 : core 0 : DM_CONF7
00000054 : core 0 : TM_CONF0
00000058 : core 0 : TM_CONF1
0000005C : core 0 : TM_CONF2
00000060 : core 0 : TM_CONF3
00000064 : core 0 : IM_CONF
00000068 : core 1 : DM_CONF0
0000006C : core 1 : DM_CONF1
00000070 : core 1 : DM_CONF2
00000074 : core 1 : DM_CONF3
00000078 : core 1 : DM_CONF4
0000007C : core 1 : DM_CONF5
00000080 : core 1 : DM_CONF6
00000084 : core 1 : DM_CONF7
00000088 : core 1 : TM_CONF0
0000008C : core 1 : TM_CONF1
00000090 : core 1 : TM_CONF2
00000094 : core 1 : TM_CONF3
00000098 : core 1 : IM_CONF
```

Full configuration:

```
# useful mem power state values:
# (default timing adjust pin values)
# to "off" : 0x01098
# to "chip disable (CD)" hub : 0x05098
# to "sleep" : 0x09098
# to "sleep trans" : 0x0d098
# to "on" : 0x11098
# to "FSM control" : 0x15098
#
# allowed transitions
# off <=> CD
# on <=>
# sleep_trans <=> CD
# sleep_trans <=> sleep
#
# e.g.:
# power on:
#   off -> CD -> on
# power off:
#   on -> CD -> off
# powered -> retention:
#   on -> CD -> sleep_trans -> sleep
# retention -> powered:
#   sleep -> sleep_trans -> CD -> on
```

Note: this is now done by
femtodriver, only needed banks are
powered

Complete APB addr space (does not include SPI regs)

#####

system-wide registers

#####

00000000 : VERSION

00000004 : RST

00000008 : DM_TIMERS

0000000C : TM_TIMERS

00000010 : IM_TIMERS

00000014 : REG_DP_ACK_DELAY

#####

per-core conf registers

#####

00000034 : core 0 : DM_CONF0

00000038 : core 0 : DM_CONF1

0000003C : core 0 : DM_CONF2

00000040 : core 0 : DM_CONF3

00000044 : core 0 : DM_CONF4

00000048 : core 0 : DM_CONF5

0000004C : core 0 : DM_CONF6

00000050 : core 0 : DM_CONF7

00000054 : core 0 : TM_CONF0

00000058 : core 0 : TM_CONF1

0000005C : core 0 : TM_CONF2

00000060 : core 0 : TM_CONF3

00000064 : core 0 : IM_CONF

00000068 : core 1 : DM_CONF0

0000006C : core 1 : DM_CONF1

00000070 : core 1 : DM_CONF2

00000074 : core 1 : DM_CONF3

00000078 : core 1 : DM_CONF4

0000007C : core 1 : DM_CONF5

00000080 : core 1 : DM_CONF6

00000084 : core 1 : DM_CONF7

00000088 : core 1 : TM_CONF0

0000008C : core 1 : TM_CONF1

00000090 : core 1 : TM_CONF2

00000094 : core 1 : TM_CONF3

00000098 : core 1 : IM_CONF

Complete APB addr space (does not include SPI regs) cont'd

#####

core 0 primary memory

#####

00100000 : DM bank 0 start
0010FFF8 : DM bank 0 end
00110000 : DM bank 1 start
0011FFF8 : DM bank 1 end
00120000 : DM bank 2 start
0012FFF8 : DM bank 2 end
00130000 : DM bank 3 start
0013FFF8 : DM bank 3 end
00140000 : DM bank 4 start
0014FFF8 : DM bank 4 end
00150000 : DM bank 5 start
0015FFF8 : DM bank 5 end
00160000 : DM bank 6 start
0016FFF8 : DM bank 6 end
00170000 : DM bank 7 start
0017FFF8 : DM bank 7 end
00180000 : TM bank 0 start
00183FF8 : TM bank 0 end
00190000 : TM bank 1 start
00193FF8 : TM bank 1 end
001A0000 : TM bank 2 start
001A3FF8 : TM bank 2 end
001B0000 : TM bank 3 start
001B3FF8 : TM bank 3 end
001C0000 : SB bank 0 start
001C01F8 : SB bank 0 end
001D0000 : RQ bank 0 start
001D01F8 : RQ bank 0 end
001E0000 : PB bank 0 start
001E01F8 : PB bank 0 end
001F0000 : IM bank 0 start
001F7FF8 : IM bank 0 end

#####

core 1 primary memory

#####

00200000 : DM bank 0 start
0020FFF8 : DM bank 0 end
00210000 : DM bank 1 start
0021FFF8 : DM bank 1 end
00220000 : DM bank 2 start
0022FFF8 : DM bank 2 end
00230000 : DM bank 3 start
0023FFF8 : DM bank 3 end
00240000 : DM bank 4 start
0024FFF8 : DM bank 4 end
00250000 : DM bank 5 start
0025FFF8 : DM bank 5 end
00260000 : DM bank 6 start
0026FFF8 : DM bank 6 end
00270000 : DM bank 7 start
0027FFF8 : DM bank 7 end
00280000 : TM bank 0 start
00283FF8 : TM bank 0 end
00290000 : TM bank 1 start
00293FF8 : TM bank 1 end
002A0000 : TM bank 2 start
002A3FF8 : TM bank 2 end
002B0000 : TM bank 3 start
002B3FF8 : TM bank 3 end
002C0000 : SB bank 0 start
002C01F8 : SB bank 0 end
002D0000 : RQ bank 0 start
002D01F8 : RQ bank 0 end
002E0000 : PB bank 0 start
002E01F8 : PB bank 0 end
002F0000 : IM bank 0 start
002F7FF8 : IM bank 0 end

EVK PCB Information

