

SPU-001 Integration Guide v0.6

This guide contains information about designing the Femtosense SPU-001 into your system¹. The intended audience is hardware and firmware engineers. Information about AI model development and compilation is not included, and is contained in other documents.

The Femtosense Sparse Processing Unit (SPU) is a neural network accelerator with 1 MB of on-board SRAM to store activations and parameters. The SPU functions as a coprocessor to a host microcontroller unit (MCU) or other System-on-Chip (SoC) processor. The host loads the parameters to the SPU once, then streams the data to be processed through the SPU in real-time. All neural network processing is handled by the SPU.

By leveraging sparsity in model development and compilation, models that would typically require several times the raw storage capacity available on the SPU can be compressed to fit.

Sections 1 and 2 can be used as a quick start guide if you are using the Femtosense API. Section 3 discusses SPI register programming. Sections 4 and 5 discuss latency and power optimizations techniques. Section 6 discusses the SPU-001 evaluation board (available by request).

¹ This document replaces the previous document “SPU TC2 Firmware Guide”. Migrating projects from SPU-001 Test Chip 2 (TC2) released in 2023 to the mass-production SPU-001 chip released in 2024 is covered in a separate document “Migrating From TC2 to the Mass Production SPU-001” (available by request).

Table of Contents

1. Hardware Integration.....	3
Requirements.....	3
Power Supplies and IO.....	3
Chip Pinout.....	4
PCB Land Pattern.....	5
PCB Layout Guidelines.....	6
Full Fanout.....	6
Limited Fanout (No Asynchronous Interrupt).....	6
Clock Input.....	7
Typical Application Circuit.....	7
Example Crystal Part Numbers.....	8
Non-volatile Model Storage.....	8
Example Flash Memory Chips ($\geq 1\text{MByte}$).....	9
2. Femtosense API.....	9
Setting up the API.....	9
API Functions.....	10
Programming Files and Basic Control Flow.....	11
3. SPI Programming and Register Map.....	11
SPI Format.....	11
SPI Write Sequence.....	12
SPI Read Sequence.....	12
Instruction Format.....	12
Router Data Format.....	13
PLL Multiplier/Divider Equation.....	14
Register Map.....	15
SRAM Programming.....	16
4. Low-power Optimization.....	17
Minimize Clock Speed.....	17
Disable Unused Cores.....	18
Use Memory FSM Mode.....	18
Turn Off the PLL When Idle (Retention Mode).....	18
5. AINR Latency Optimization.....	19
Components of End-to-End Latency.....	19
Latency Example for Femtosense EVK2.....	19
6. Evaluation Board.....	20
Schematic.....	21
Pinout.....	22
7. Change Log.....	23

1. Hardware Integration

Requirements

The SPU acts as a neural network inference co-processor. The only data interface is via SPI and an interrupt pin. It does not have its own audio interface or GPIO, so the following system components are also required:

- **Host processor:** A 1.8-3.3V IO MCU or SoC that supports SPI communication $\geq 4\text{MHz}$. Optionally, the SPU also provides an asynchronous interrupt signal that can be connected to one of the host's GPIO pins.
- **Audio interface:** The SPU supports 4-bit weights and 8-bit activations, or 8-bit weights and 16-bit activations. Depending on the model compute time, different sample rates and data frame sizes may be supported.
- **Non-volatile model storage:** This should be at least as large as your model parameters (up to 1MByte, if your model uses all SPU resources). Larger storage is recommended to account for file system overhead and in case multiple models need to be stored simultaneously. If this storage is not available, the model can also be streamed from an external source such as USB, Bluetooth, or Wi-Fi.

Power Supplies and IO

The SPU is available in a 15-pin CSP package. There are 4 different power supply pins:

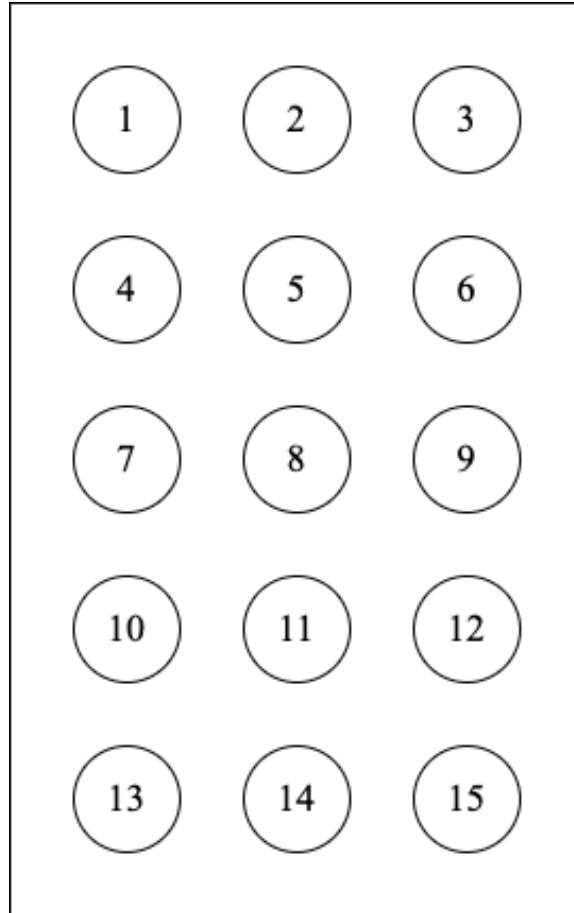
- **DVDD (1.8 - 3.3V):** This should be the same voltage as your MCU IO level
- **VDDM (0.8V):** SRAM memory power
- **VDD (0.8V):** Nominal core power
- **VDDA (0.8V):** PLL power

Serial data communication is done using standard 4-pin SPI. Additionally, the SPU provides an interrupt signal that can tell the host MCU when an output data frame is ready for reading, but this is optional if desired.

Chip Pinout

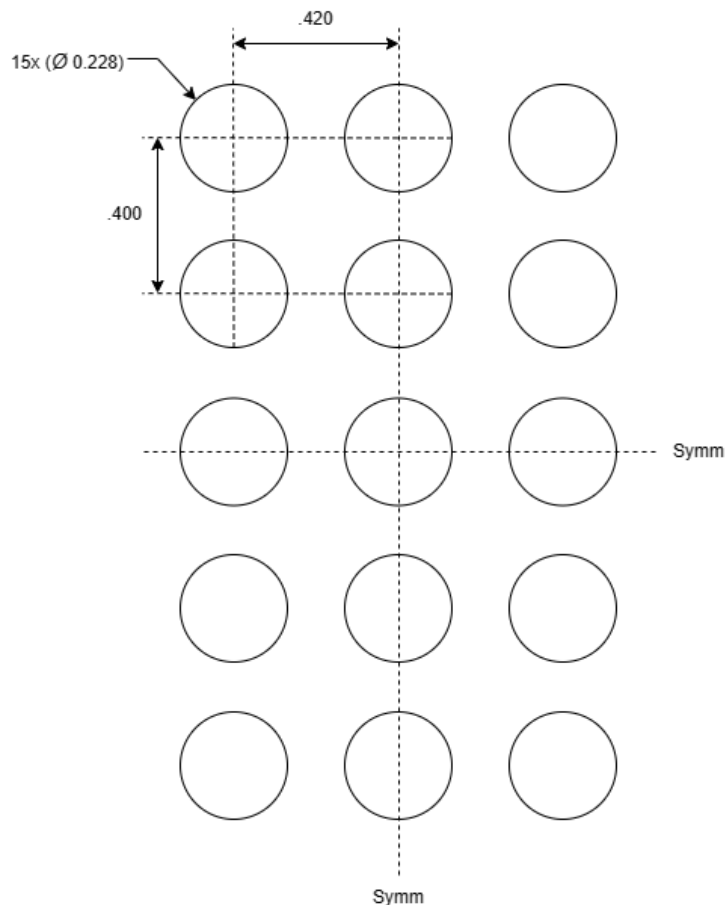
The pinout of the SPU is shown below (view from the top looking “through the chip”, or the pinout applied to the PCB).

PIN #	ID
1	SPI_MISO
2	VDD
3	VSS
4	SPI_SCK
5	INT
6	DVDD
7	SPI_MOSI
8	VDDM
9	RST
10	SPI_CS
11	VDDA
12	OSC_PADI
13	VSS
14	VDD
15	OSC_PADO

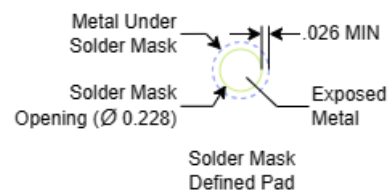


PCB Land Pattern

An example PCB land pattern is recommended below. The exposed pad size is recommended to be 228 μ m. A solder mask defined (SMD) pad design is recommended on the right. All dimensions are in mm.



Land Pattern Example
(Exposed Metal Shown)

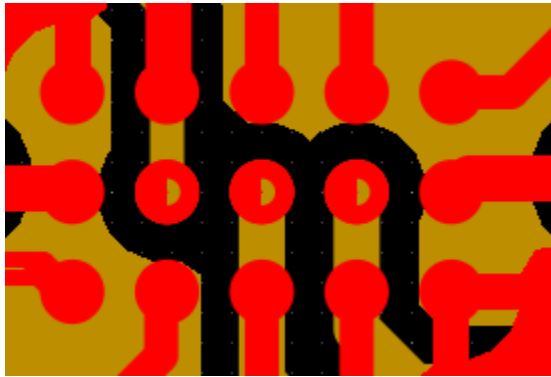


PCB Layout Guidelines

There are two layout configurations, one for full fanout including the interrupt signal and one for limited fanout that does not use the interrupt but is easier to route.

Full Fanout

If your PCB technology allows, the 3 interior pads should be fanned out so that the INT signal can be accessed by the host MCU, and the VDDA and VDDM signals can be decoupled separately from VDD. Using the recommended SMD land pattern dimensions, a 280µm via-in-pad can be used to escape these 3 interior pads as shown in the example below:

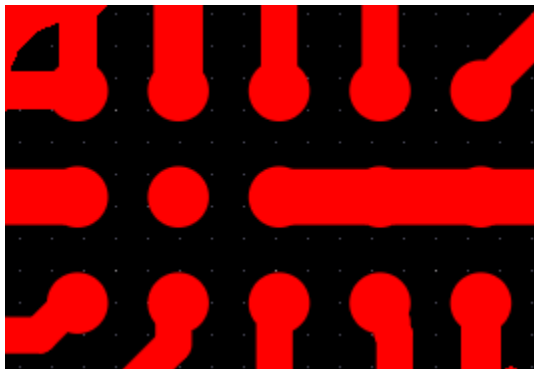


Example PCB layout with full fanout (interior pads escaped). Red=L1 Copper, brown=L2 Copper. The 3x dual-color circles indicate via holes between L1 to L2. Pin 1 is in the bottom-left.

Both VDD and VSS pins should be connected.

Limited Fanout (No Asynchronous Interrupt)

Optionally, if appropriate via-in-pad technology is not available, pins 8 (VDDM), 11 (VDDA), and 14 (VDD) can be routed together, then pin 5 (INT) should be disconnected as shown in the example below:



Example PCB layout with limited fanout. Red=L1 Copper. Pin 1 is in the bottom-left.

Do not route the 0.8V from pin 2 to pin 8 through pin 5, as pin 5 can generate IO level voltage ($>0.8\text{V}$).

Both VDD and VSS pins should be connected.

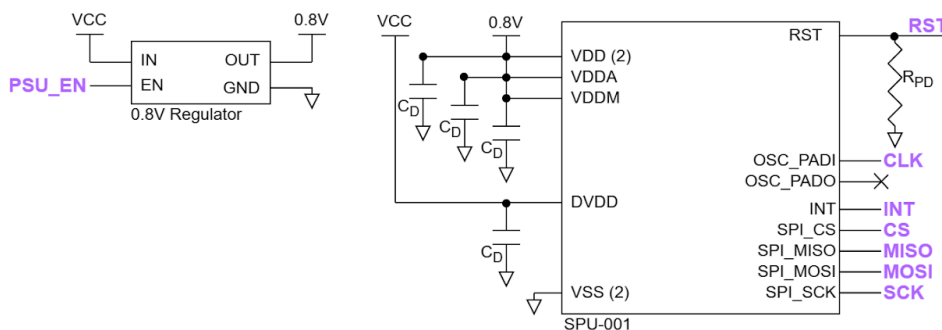
If using this limited fanout routing, the asynchronous INT signal will not be accessible by the host MCU. The INT pin is an asynchronous signal that indicates when SPU inference data is waiting to be read back (i.e. each time the model running on the SPU is done processing one set of inputs). If this asynchronous INT signal is disconnected, the INT signal can alternatively be read by polling an SPI register.

Clock Input

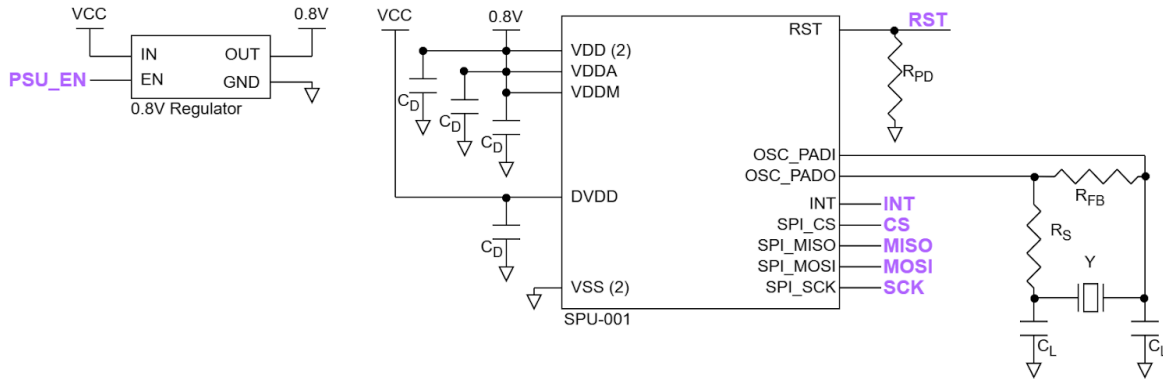
The SPU requires a reference clock for its Internal PLL. This can be an externally generated clock (full swing digital signal at IO voltage level), or a 32.768KHz crystal oscillator excited by the SPU's built-in oscillator pad (as shown in the next section). The PLL can then be configured to output a multiplied system clock to the cores.

Typical Application Circuit

For an external clock reference, the following schematic shows a typical application circuit. **Purple** signals represent SPU control IO from adjacent systems (e.g. host MCU). **PSU_EN** is optional, and only required if your system is sensitive to boot-up power consumption so that the SPU boot sequence can be precisely controlled. Ideally, the SPU's power rails are brought up with the reset held. This could help prevent an indeterminate state (with indeterminate power consumption) if the SPU is powered up while not under reset.



If using the limited fanout layout in the previous section, only one C_D is needed for the combined VDD/VDDA/VDDM connection (in addition the C_D on DVDD), and INT can be disconnected. For a crystal oscillator reference, the schematic below shows a typical application circuit. Limited fanout routing can also be used with this circuit.



The following values are used in the schematics:

Symbol	Value	Comment
C_D	100nF	+/-20%
C_L	22pF	+/-1%
Y	32.7680KHZ	12.5pF load, 70K Ω ESR
R_{FB}	4.7M Ω	+/-1%
R_S	1 Ω	+/-1%
R_{PD}	100K Ω	Optional
VCC	IO Voltage (host voltage)	0.8V - 3.3V

Femtose sense can provide additional details about alternative crystals or reference clocking architectures not included above. Example crystal parts are given below:

Example Crystal Part Numbers

Manufacturer	Part Number	Size	Temperature Rating
Abracon LLC	ABS07-32.768KHZ-T	3.20mm x 1.50mm	-40°C ~ 85°C
Micro Crystal	CM8V-T1A-32.768KHZ-12.5PF-20PPM-TB-QC	2.0mm x 1.2mm	-40°C ~ 125°C

Non-volatile Model Storage

The SPU stores its model parameters in volatile SRAM, so it must be programmed after power up. Since SPU SRAM size is 1MByte, we recommend at least 1MByte of dedicated non-volatile

storage in your system in order to store model parameters though less is acceptable if planned application sizes are well below 1MByte. This can be in the MCU flash, external flash, or similar non-volatile storage. Parameters could also be streamed over a bluetooth or wifi radio through the host if the device is connected to another system in this way. The following table shows a list of example low-power SPI Flash memory chips with the same IO voltage range as the SPU, but any non-volatile storage of similar size and IO voltage will work.

Example Flash Memory Chips (≥ 1 MByte)

Manufacturer	Part Number	Package/Size	Capacity	IO/Supply Voltage
Macronix	MX25R8035FBDIL0	WLCSP / 1.52mm x 1.25mm	1 MByte	1.65V ~ 3.6V
Macronix	MX25R8035FZUIL0	USON / 2.0mm x 3.0mm	1 MByte	1.65V ~ 3.6V
Macronix	MX25R1635FBDIL0	WLCSP / 1.99mm x 1.58mm	2 MBytes	1.65V ~ 3.6V
Macronix	MX25R1635FZUIL0	USON / 2.0mm x 3.0mm	2 MBytes	1.65V ~ 3.6V
Macronix	MX25R3235FZBIL0	USON / 4.0mm x 3.0mm	4 MBytes	1.65V ~ 3.6V

2. Femtosense API

Setting up the API

First, the following simple wrapper functions in **spu001.c** should be implemented in order to connect the API to your hardware:

- SpuInterruptRead()
- SpuDelayUs()
- SpuResetEnable()
- SpuResetDisable()
- SpuSpiEnable()
- SpuSpiDisable()
- SpuSpiWrite()
- SpuSpiRead()

SpuInterruptRead() is only required if the asynchronous interrupt pin is used. Once these are implemented, the following functions can be used to control the SPU.

API Functions

FemtoseNSE provides an C API² to communicate with the SPU001. These functions are included in **spu001.c** and **spu001.h**, and cover the main functionalities as described below:

- **SpuInit()**: Initializes the SPU, including power-up and reset sequence. It checks that the SPU is connected properly, and should be called first before any of the other functions.
- **SpuWriteModelValue()** and **SpuWriteModelBatch()**: Writes model and memory configuration parameters from non-volatile storage to the SPU's SRAM.
- **SpuWriteData()**: Writes one frame of input data (e.g. audio, sensor data) for the SPU to process.
- **SpuReadData()**: Reads the result of one frame of input data. This should not be called before checking that the interrupt value == 1 indicating that SPU output data is available.
- **SpuEnableCores()**: Enables or disables individual SPU cores via clock gating. This is useful if your model does not need both cores, or if a particular core can be turned off to save power.
- **SpuPollInterruptPin()**: Polls the interrupt signal using the SPU interrupt pin.
- **SpuPollInterruptRegister()**: Polls the interrupt signal using the SPU SPI register instead of the interrupt pin.
- **SpuClockSetOutputDivider()**: Set PLL output divider. Allowed values are 1 or positive even integers less than or equal to 16. Do not change the output divider unless the SPU is idle.
- **SpuDisablePll()** and **SpuEnablePll()**: Enter and exit retention mode where pll is disabled in order to save power when not being used for an extended period of time.

Before initializing the SPU with **SpuInit()**, create a **SpuConfiguration** variable and initialize it with the values that reflect the hardware configuration:

- **Clock_type**: Either **kSpuExtClk** indicating that an external clock is used, or **kSpuOsc** indicating that the SPU-controlled crystal oscillator is used.
- **Crystal_frequency_hz**: The frequency of the PLL input clock or SPU-controlled crystal oscillator (if used).
- **Core_clock_frequency_mhz**: The PLL output frequency before the output divider (or when the output divider is set to 1). For smaller algorithms such as Wake Word Detection (WWD), Google Speech Commands (GSC), Spoken Language Understanding (SLU), and power-optimized AI Noise Reduction (AINR), 100MHz is a good place to start. For larger algorithms such as latency-optimized AINR, 250MHz is a good place to start. Information about selecting an optimal frequency is discussed in Section 4.
- **Spi_clock_frequency_mhz**: The SPI clock frequency of the host processor. Do not set this > 50.

² Distributed within the source code for EVK2 and EVK2v2 on the release portal

Programming Files and Basic Control Flow

In order to use the SPU, you first need compiled programming files that contain address/data pairs of your model parameters. Once this is loaded in your non-volatile model storage, the basic control flow required to use the SPU is as follows:

1. Call **SpuInit()** with the **SpuConfiguration** matching your hardware. An error will be returned if initialization fails. Use **SpuResultToString()** to get a description of the error.
2. Load the SPU with your model parameters by calling **SpuWriteModelValue()** for every address/data pair in your programming file. If address writes are sequential (the address locations differ by 4), **SpuWriteModelBatch()** can be used to write one stream of data to a sequence of addresses in order to save overhead.
3. For every frame of data, send the data frame as an array of 16-bit integers using **SpuWriteData()**.
4. Wait for the SPU interrupt to equal 1. The SPU interrupt can be read from the interrupt pin using the host's GPIO, or by calling **SpuPollInterruptPin()**. The same signal can also be read via SPU by calling **SpuPollInterruptRegister()**.
5. Read the processed data frame (result) from the SPU using **SpuReadData()**.
6. Repeat steps 3-5 for every frame of data.
7. If the SPU will not be used for an extended period of time (e.g., the "AI feature" of the product is turned off), enter retention mode by calling **SpuDisablePII()** to save power. Return from retention mode by calling **SpuEnablePII()**.

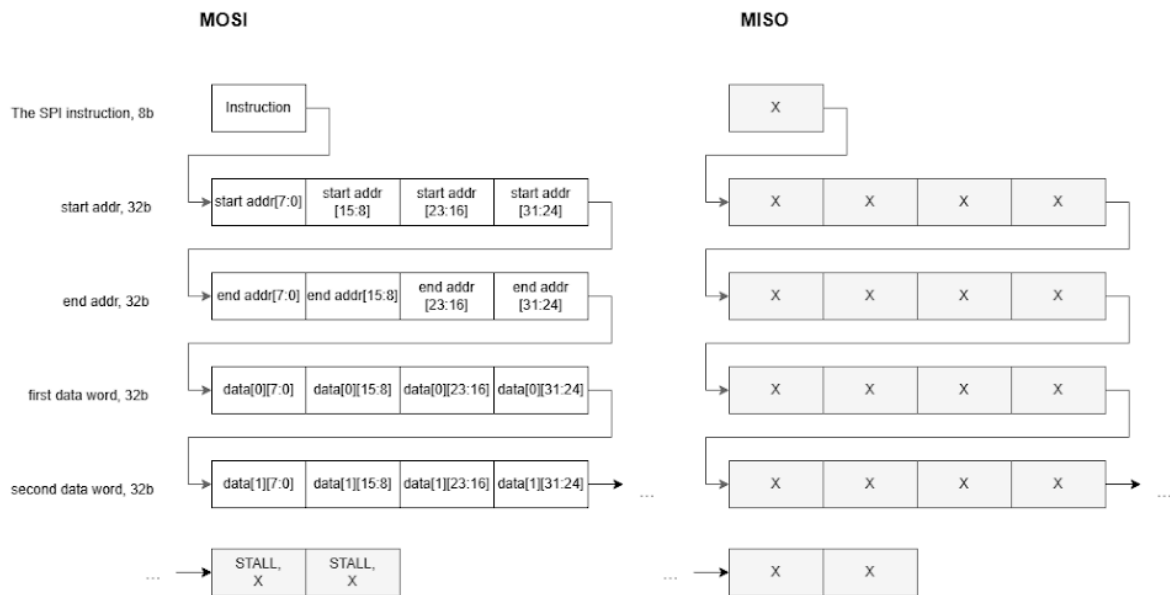
3. SPI Programming and Register Map

If you do not use the Femtosense API, you can program the SPU by directly manipulating registers.

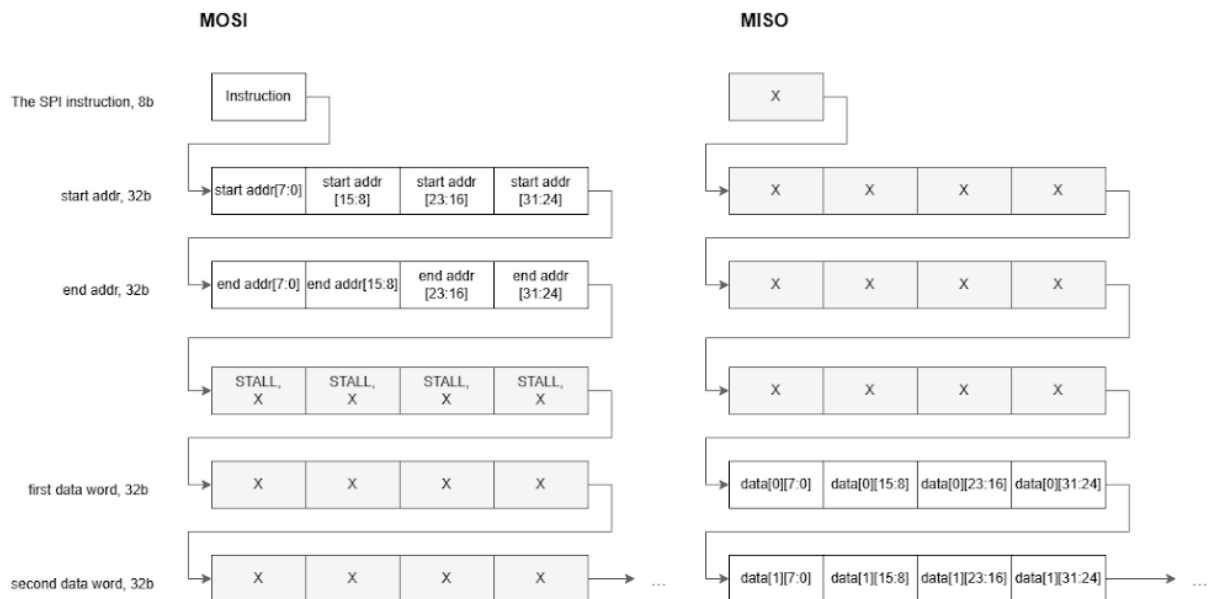
SPI Format

The SPU communicates using SPI MODE0 with clock speeds 4MHz - 50MHz. The following sections show the read and write formats.

SPI Write Sequence



SPI Read Sequence



Instruction Format

The format of the instruction byte is shown below:

MSB		LSB
X	Inst Type 2b	wr_en 1b

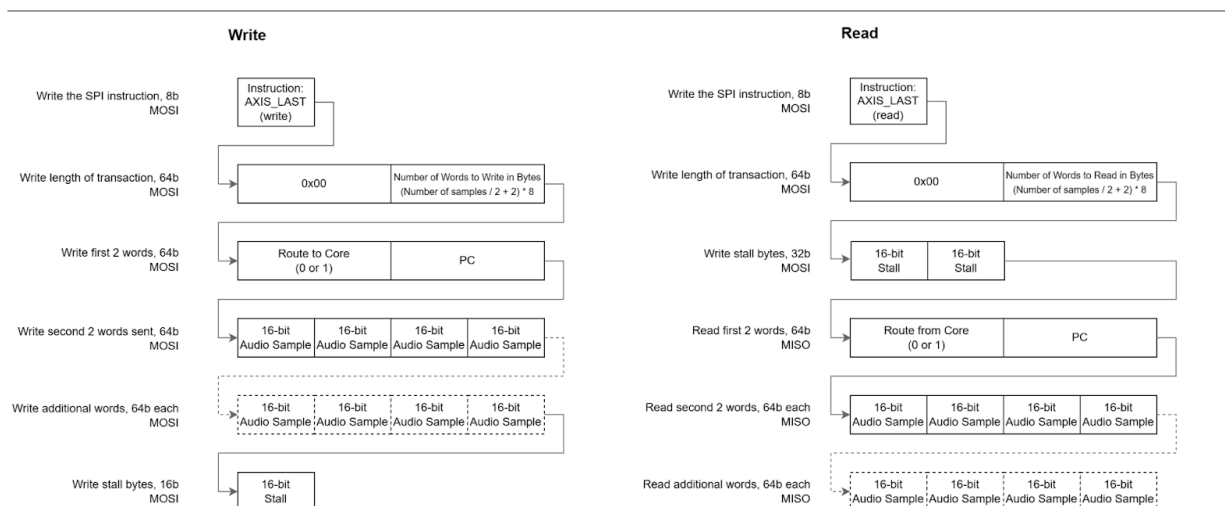
Where wr_en=1 for writing data and wr_en=0 for reading data. Instruction types (Inst Type) are given in the table below:

Instruction Type	Value (2b)	Description
IOREG	0x00	Read/write to registers listed in the Register Map below
APB	0x01	Read/write to the APB address space (e.g. SRAM)
AXIS	0x02	Read/write data to the SPU Router (input/output data to model). For writes, this instruction is used to break up an input into multiple SPI transactions. AXIS_LAST must be used as the last message. E.g. a multi-part message could be as four transactions: AXIS, AXIS, AXIS, AXIS_LAST. However, the header "Route to Core" and "PC" should only be included in the first message.
AXIS_LAST	0x03	Read/write the last data to the SPU Router. Processing will begin after the final byte is written with this instruction. Can be used for a single complete input message.

Router Data Format

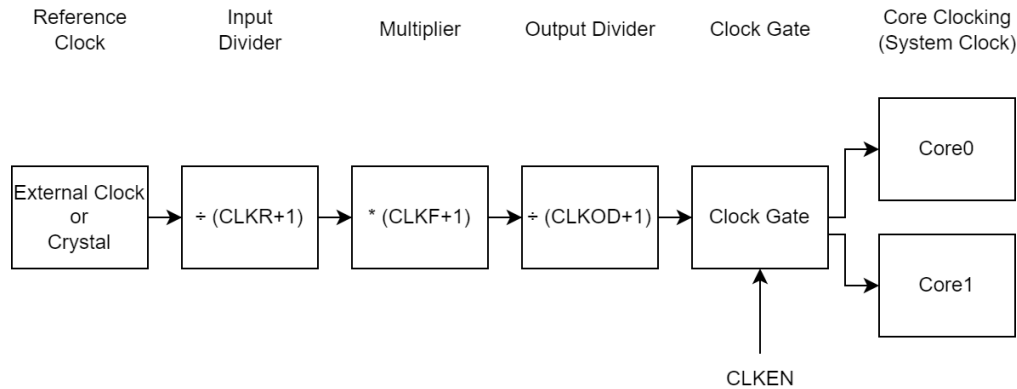
Data frames sent to the SPU for processing use the AXIS/AXIS_LAST instructions using the format shown below. This is the same format as the SPI read/write, but includes a header in the first two words before the samples are read/written.

AXIS Example



PLL Multiplier/Divider Equation

The PLL multipliers and dividers are illustrated in the diagram and equation below:



$$\text{System Clock Frequency} = \text{Reference Clock} * \frac{(CLKF + 1)}{(CLKR + 1)(CLKOD + 1)}$$

The PLL Bandwidth factor is also required when setting up the PLL:

$$BWADJ = \frac{(CLKF + 1)}{2} - 1$$

For normal operation, the clock gate can be set to CLK_EN=1.

Register Map

A map of the SPI registers is given below. These can be accessed using the IOREG instruction (0x00). Each register is 32-bits wide.

SPI Address Location	Name	Register Description	Bit	Bit Name	Description	Default
0x00	SPI_CONF_IDX	SPI Conf	11	CORE_RESET	Core Reset (1=reset)	1
			10	<unused>	<unused>	0
			9	<unused>	<unused>	0
			8	<unused>	<unused>	0
			7	<unused>	<unused>	0
			6	<unused>	<unused>	0
			5	<unused>	<unused>	0
			4	<unused>	<unused>	0
			3	<unused>	<unused>	0
			2	DISABLE_CORE_1	Disable Core1	1
			1	DISABLE_CORE_0	Disable Core0	1
			0	NO_HALF_CYCLE_DELAY	SPI mode (0 or 1)	0
0x04	IO_CONF_IDX	IO Conf (pad conf)	11	INT_EN	Interrupt Enable (1=enabled)	0
			10	OSC_EN	oscillator enable	0
			9	OSC_TEST_EN	oscillator test mode enable	1
			8	FAST_SLEW_EN	pad fast slew mode enable	0
			7:6	DRIVE_STRENGTH	pad drive strength (higher value = more)	0x01
			5	PULL_EN	weak pull enable	0
			4	PULL_DIRECTION	<unused>	0
			3	PAD_RETENTION_ON	PVSENSE input pull_direction, pads will hold last value when not powered	1
			2	OSC_SF1	osc_SF1 crystal frequency range configuration	0
			1	OSC_SF0	osc_SF0 crystal	0

					frequency range configuration	
			0	RET_ON	pad retention enable	0
			6	CLK_EN	enable PLL output clock gate	0
			5	PWRDN	power down pll (1=powered down)	1
			4	BYPASS	bypass reference clock through PLL	1
			3	TEST	PLL test enable	0
			2	FASTEN	fast pll turn on enable	0
			1	ENSAT	Enable PLL saturation behavior (do not change)	0
0x08	PLL_CONF_WORD0	Misc PLL Control	0	RESET	reset pll	1
0x0C	PLL_CONF_WORD1	PLL BW ADJ	11:0	BWADJ	PLL bandwidth value	0x1F3
0x10	PLL_CONF_WORD2	CLKOD	3:0	CLKOD	PLL output divider value	0x01
0x14	PLL_CONF_WORD3	CLKF	12:0	CLKF	PLL clock multiplier value	0x3E7
0x18	PLL_CONF_WORD4	CLKR	5:0	CLKR	PLL input divider value	0x00
0x1C	PLL_LOCK	PLL_LOCK	0	PLL_LOCK	PLL Lock indicator	0x00
0x20	<unused>	<unused>				
0x24	INTERRUPT	Interrupt Read	0	INT	Interrupt Value (0 = not interrupt, 1 = interrupt)	0x00

SRAM Programming

A compiled SPU model will generate the files **0PROG_A** and **0PROG_D** that contain the data that should be loaded into the SPU's SRAM to enable the model. A snippet of these files is shown below:

OPROG_A		OPROG_D	
1	00000064	1	00005098
2	00000064	2	00011098
3	00000034	3	00005098
4	00000034	4	00011098
5	00000038	5	00005098
6	00000038	6	00011098
7	0000003c	7	00005098
8	0000003c	8	00011098
9	00000040	9	00005098
10	00000040	10	00011098
11	00000044	11	00005098
12	00000044	12	00011098
13	00000048	13	00005098

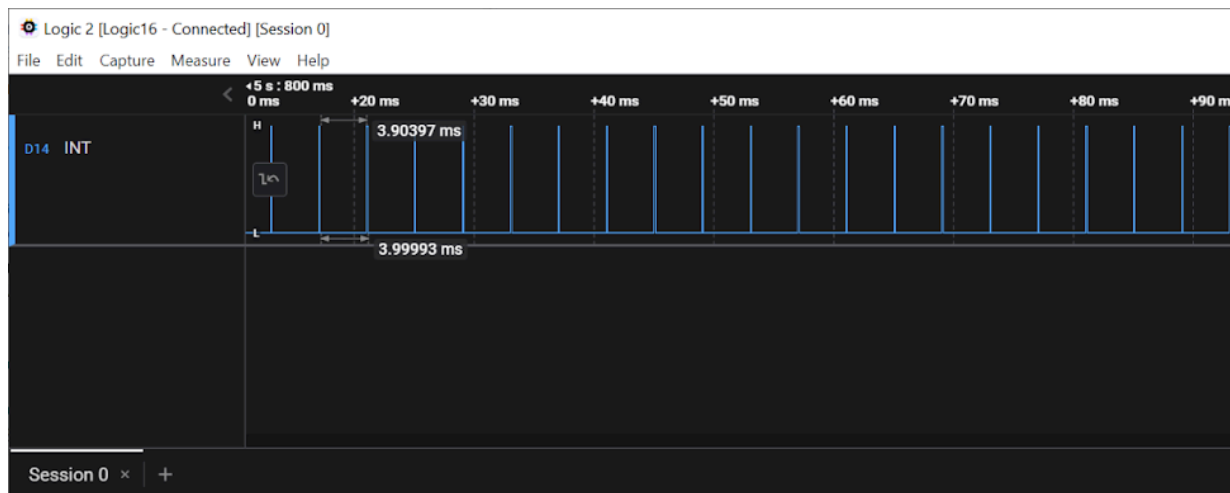
Each line of **OPROG_D** contains the value that should be written to the corresponding memory location in **OPROG_A**. For example, to write the first line in the snippet above, the value **0x00005098** should be written to location **0x00000064**. Every line in these files should be written to the SPU. Write using the instruction 0x01 (APB) or **SpuWriteModelValue()** in the API.

4. Low-power Optimization

In order to minimize the SPU's power consumption, the following optional optimizations may be implemented in firmware. If these are not done, the SPU will still function correctly, but will consume more power than necessary.

Minimize Clock Speed

The clock speed seen by the cores (after the PLL output divider) should be minimized to be as slow as possible while still allowing for the SPU to complete processing before the next data frame. This increases compute latency by up to the duration of one frame in order to save power. If your algorithm has an audio output, you can check that the SPU has completed processing in time by listening to the audio output and checking to make sure that no frames are skipped because the SPU is taking too long to process. Alternatively, you can make sure that the data processing is completed in time by checking the timing of the interrupt signal. The interrupt signal should fire exactly once per frame. For example, for a 4ms data frame, the interrupt signal should look like the screenshot below, firing exactly every 4ms. If the time between interrupt pulses is longer than the frame duration, the clock speed should be increased.



Although the clock speed should be minimized, it is important that the PLL output frequency before the output divider (set with **core_clock_frequency_mhz** in the API) is always $\geq 30\text{MHz}$ in order to maintain stability. It is also important that during data IO, the PLL output frequency after the clock divider is ≥ 2 times the SPI SCK frequency during this time.

The output divider can be set with **SpuClockSetOutputDivider()** in the API, however it should not be changed unless the cores are idle. Note also that the output divider can only be set to 1 or an even positive number less than or equal to 16.

Disable Unused Cores

Some algorithms can be compiled to only use Core0. In this case, Core1 can be disabled with **SpuEnableCores()** in the API.

Use Memory FSM Mode

During memory configuration (typically included in first lines of the compiled programming file), the memories can be configured to use internal FSM control for their power states. In this mode, the memories will sleep automatically after a preprogrammed delay. This typically saves some power. During programming, the normal memory configuration routine will write **0x11098** to the memory configuration register ($0x34 \leq \text{APB address} \leq 0x98$). To enable FSM control, write **0x15098** instead of **0x11098** to the memory configuration register.

Turn Off the PLL When Idle (Retention Mode)

When the SPU does not need to process data for an extended period of time (i.e., when the algorithm is turned off), it can be placed in retention mode by turning off the PLL in order to minimize power consumption. In this mode, the contents of the SPU's SRAM are retained so

that the SPU can be quickly reactivated when needed. In the API, enter retention mode with **SpuDisablePll()**, and reactivate the SPU with **SpuEnablePll()**.

5. AINR Latency Optimization

AINR systems are latency sensitive since the user will expect to hear the noise-reduced audio at the same time they experience it visually. If the latency is too high, the user will also hear an echo of their own voice in the output of the algorithm. If optimizing for overall latency, it is important to understand the three places where latency is generated:

Components of End-to-End Latency

- **Algorithm Latency:** Latency inherent to the algorithm due to the overlap-add associated with the Short-Time Fourier Transform and its inverse. Each audio frame received by the SPU is processed alongside the previous input frame. This step generates an output audio frame corresponding to the previous input frame. The resulting latency is 2x the hop size, where the hop size is the time between audio frames.
- **Compute Latency:** Time taken by the SPU to compute the algorithm. This is at most 1x the hop size, as the algorithm must complete once per audio frame to keep up with the audio stream. This can be controlled by changing the SPU's clock speed.
- **Additional System Latency:** Host buffering in the microcontroller, codec, analog-to-digital converter, and digital-to-analog converter, if any. For example, on the Femtosense Evaluation Kit 2 (EVK2), the host consists of a [Teensy 4.1](#) microcontroller and [TI TLC320AIC3206](#) audio codec. After firmware implementation, this latency is:
17 sample ADC queue + 21 sample DAC queue + 1 hop host audio queue

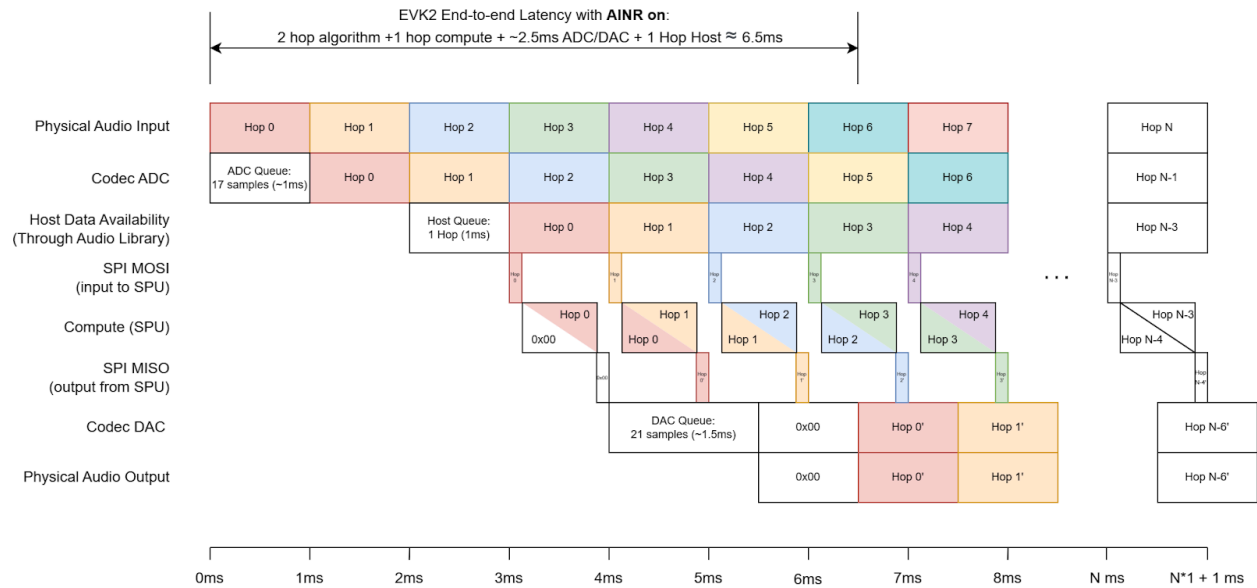
Summing these parts results in the **End-to-end Latency**. In order to minimize latency, you should optimize your system to reduce these three components.

Latency Example for Femtosense EVK2

As an example, when using 16KHz audio and a 1ms hop AINR algorithm on the EVK2, the End-to-end Latency is then:

$$2ms \text{ algorithm} + 1ms \text{ compute} + (\sim 1ms \text{ ADC queue} + \sim 1.5ms \text{ DAC queue} + 1ms \text{ host queue}) \\ \approx 6.5ms$$

The figure below details the latency of the audio samples as they are processed:

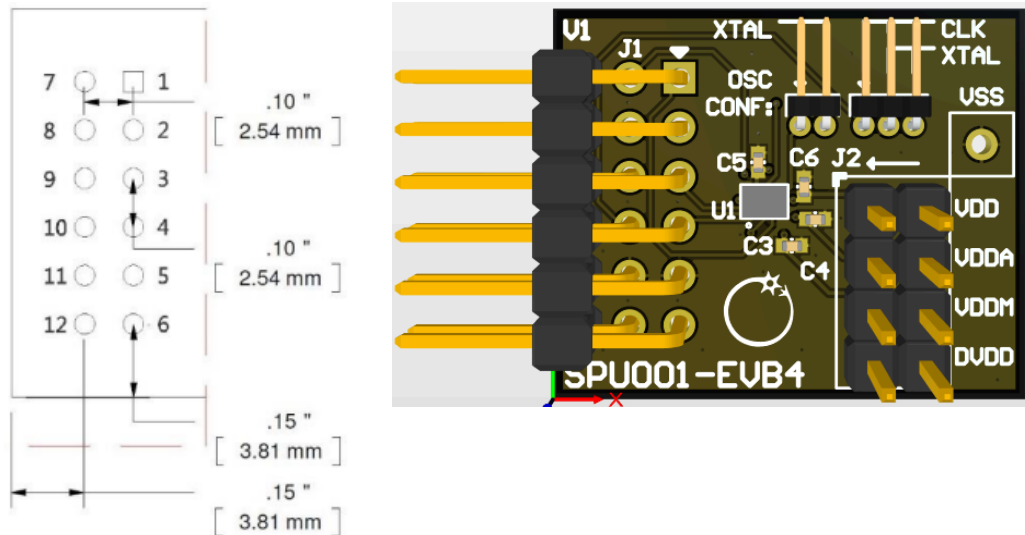


6. Evaluation Board

In addition to evaluation kits (EVKs), Femtosense also provides an evaluation board for the mass-production SPU called Evaluation Board 4 (EVB4). The schematic for this board is shown below. This board connects to the host processor through J1's PMOD-like interface ([PMOD type 2A](#)), and can be configured to use either an external IO-level reference clock TEST_CLK, or the crystal oscillator circuit near Y1, using J3 and J4.

Pinout

The pinout of the main header J1 is pin-compatible with the Digilent PMOD interface type 2A specification:



Pin	Description	Note
1	SPI Chip Select for SPU-001	active low
2	SPI MOSI signal	Mode 0
3	SPI MISO signal	Mode 0
4	SPI SCK clock signal	Mode 0
5	Ground	Mode 0
6	PSU_EN*	Enable 0.8V regulator (active high)
7	SPU Interrupt signal	logic high when data frame is ready
8	SPU Reset signal	active high
9	SPI Chip Select for onboard flash chip	PN: Macronix MX25R3235FZBILO
10	Reference Clock for SPU	VCC level
11	Ground	
12	VCC	1.8V-3.3V

*For standard PMOD type-2A, PSU_EN = HIGH.

SPU current consumption on each of its rails can be measured with an ammeter on the labeled rows of J2.

7. Change Log

Version	Release Date	Description
0.1 (Provisional)	2024-02-05	Initial release replacing “SPU TC2 Firmware Guide”
0.2 (Provisional)	2024-02-20	Added more PCB specifications
0.3 (Provisional)	2024-03-08	Added example Flash, crystal, 0PROG format, and API setup instructions.
0.4	2024-03-20	Corrected router data format chart.
0.5	2024-04-22	Text updates
0.6	2024-04-29	Added more information about low-power-optimization