



IIS3DWB10IS: ISPU (intelligent sensor processing unit)

Introduction

This document provides usage information and application hints related to the ISPU (intelligent sensor processing unit) in ST's IIS3DWB10IS.

1 ISPU

The ISPU (intelligent sensor processing unit) is an embedded programmable core that allows reading sensor data and processing it inside the IIS3DWB10IS device and can directly provide, when necessary, the results of said processing to an external microcontroller. The ISPU can run any type of processing, from basic signal processing to artificial intelligence algorithms.

The ISPU in the IIS3DWB10IS device is based on the STRED architecture, a proprietary architecture developed by STMicroelectronics, targeting extremely low power consumption and a small silicon area.

In particular, the ISPU architecture is a 32-bit RISC Harvard architecture and features a floating-point unit (FPU) to accelerate single precision floating-point operations (multiplication, addition, subtraction, maximum, minimum, accumulate and multiply). The complete ISPU instruction set is available in [Table 10 of Appendix A: ISPU instruction set](#). The ISPU also provides a set of hardware blocks, usable from the system registers, to accelerate loops, FFT computation, circular buffers, and square root computation. Details on these hardware blocks are available in [Section 13: Hardware accelerators](#).

The ISPU is supported by two separate RAMs:

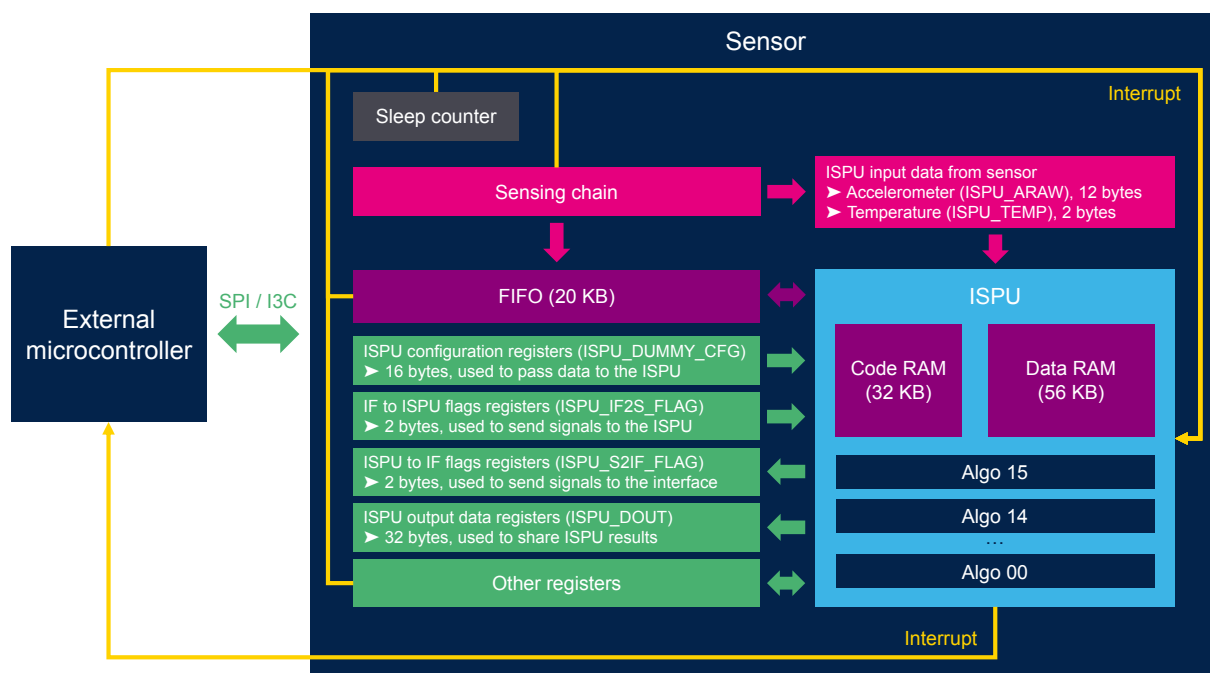
- 32 KB
- 56 KB

Code and data are normally meant to be stored separately on, respectively, the 32 KB RAM and the 56 KB RAM, allowing simultaneous fetching and improving efficiency. However, the usage of the two RAMs is flexible, and they can be used interchangeably for either code or data. The only constraint is that the interrupt vector table (IVT) must be placed at the start of the 32 KB RAM.

Note that the device does not have any on-board nonvolatile memory to store the program, which means that the program must be reloaded to the device RAM each time power to the device is removed and reapplied.

The ISPU is able to interact both with the sensor, from a set of internal and external registers and a first-in-first-out (FIFO) buffer, and with an external microcontroller, communicating through a set of registers accessible both by the ISPU and the external microcontroller over the device interface and through the FIFO buffer, where the ISPU can store data and that the external microcontroller can read.

Figure 1. Sensor with ISPU core



The ISPU is designed to be normally in a sleep state, where the clock is disabled. When new data is generated by the sensor, the ISPU is woken up and the new data can be processed.

As shown in [Figure 1](#), the ISPU can access all the sensor data generated by the device (accelerometer and temperature), either from dedicated registers, or from the FIFO, if properly configured. The ISPU can also access data written by an external microcontroller in dedicated registers. This data may represent configuration values needed by the processing logic or flags set to request an action from the ISPU. Using all this input data, the ISPU can run up to 15 algorithms and, once the data has been processed, it can write the results in the general-purpose output registers, that the external microcontroller can read to retrieve the needed information, or in the FIFO, also accessible by the external microcontroller.

The ISPU can also generate interrupt signals on the INT1 and INT2 pins of the device in order to signal to the external microcontroller that there is new relevant data to be read from the output registers or the FIFO, or even directly signal the detection of a specific event without the need to transmit any data. By using the interrupts, it is possible to keep the external microcontroller in a sleep state for most of the time in order to drastically reduce the power consumption of the system.

Once the ISPU has finished all processing for the new data and has updated the relevant output registers or pushed the results of the processing to the FIFO, then it can be put again in the sleep state to avoid consuming power while waiting for new data to process.

The ISPU can also be configured to receive interrupts from different sources, some originating from the sensor itself (new sensor data available, sleep counter reaching the threshold, FIFO full or reaching the watermark threshold) and some generated externally by the user (signal on INT2 pin or specific bit set through the device interface), allowing the user to process data or perform specific actions upon receiving such interrupts.

The ISPU program can be written in standard C code, which allows for high flexibility and reuse of code written for other architectures. It is nonetheless recommended to implement appropriate optimizations for the ISPU architecture if the performance of the processing logic is insufficient to keep up with the sensor data rate or to further reduce power consumption. Operations that are not hardware accelerated, like floating-point division or 32-bit integer multiplication, should be avoided as much as possible. The hardware blocks described in [Section 13: Hardware accelerators](#) can also be exploited to improve performance even further.

A toolchain is provided on the STMicroelectronics website ([ISPU-Toolchain](#)), containing tools for the compilation of the code (based on standard open source tools) and the conversion of binary files into a format suitable to be loaded into the device. Additionally, templates and examples that can be used to quickly start programming the ISPU, together with tutorials, firmware and additional resources, are available on the [ISPU GitHub repository](#).

Note: Support for some of the functions described in this document (for example, processing cycle, algorithm management, interrupt handling) is already available in the ISPU code of the templates and examples.

Note: This document does not describe how to use the software tools mentioned above, but describes every functionality related to the ISPU at the device level. For the tools, refer to the GitHub repository linked above.

2 Processing cycle

The ISPU is designed to work in a cycle of wake, process, sleep, and repeat, in order to process data when needed and save power when no action is required.

The main source that can wake up the ISPU is an internal signal generated by the sensor when new accelerometer data is available. The rate at which new data is generated, and thus the ISPU is woken up, depends on the value set in the ISPU_RATE_[3:0] field of the ISPU_CTRL1 (0x18) register. This setting can be configured independently from that of the ODR_XL_[3:0] field of the CTRL1 (0x10) register, which instead controls the rate of the accelerometer data made available in the device output registers and batched in the FIFO. However, the accelerometer must not be set in idle mode (ODR_XL_[3:0] = 0x0) in order for the ISPU to work.

Table 1. ISPU rate selection

ISPU_RATE_[3:0]	Rate
0000	0 Hz
0001	Reserved
0010	2.5 kHz
0011	5 kHz
0100	10 kHz
0101	20 kHz
0110	40 kHz
0111	80 kHz
1000-1111	Reserved

The signal generated when new data is available can also trigger, if so configured, a high-priority interrupt to the ISPU (see [Section 11.3: Data valid / high-priority interrupt](#)). This is particularly useful if the ISPU is already awake and executing. The ISPU stops the execution of the code, calls the interrupt service routine, where the user can store the new data for later processing, and then resumes the execution from where it was stopped.

There is also a set of low-priority interrupts (see [Section 11.5: Low-priority interrupts](#)), which can trigger an interrupt service routine and wake up the ISPU if previously asleep. These interrupts work independently from the ISPU rate configuration, meaning that the ISPU rate can be configured to zero in order to wake up the ISPU only when a low-priority interrupt is generated. While in most cases it might be needed to have access to the sensor data from the internal registers, which requires an ISPU rate setting different from zero, this could be useful, for example, to accumulate sensor data in the device FIFO, wake up the ISPU when the FIFO contains a preconfigured number of samples and retrieve the data from the FIFO for processing.

Thanks to the high-priority interrupt and the FIFO, both of which can be used to store new data generated by the sensor while the ISPU is executing, there is no hard limit on processing having to complete before a new data sample is generated. However, depending on the application, it is fundamental that the processing executes fast enough to keep up with the data it needs to process, otherwise the application will not work correctly. For this reason, methods to monitor the execution time of the ISPU processing are offered. Refer to [Section 17.1: Measuring execution time](#) for details.

Once the processing is completed, the ISPU can be put in the sleep state by attempting to perform a load from an address between 0xC0000 and 0xFFFFF (not mapped to any memory or register). This causes the clock of the ISPU to be stopped. This can be achieved, for example, using the following assembly code:

```
ldb %r0, [0xC0000]
```

It is recommended to surround the load instruction with `nop` instructions:

- The preceding `nop` ensures that if an interrupt is triggered by the previous instruction, the interrupt can be executed before the ISPU is put to sleep.
- The following `nop` is to ensure that the ISPU goes to sleep before executing the next instruction.

When a new data sample arrives from the sensor generating the trigger to wake up the ISPU, or a low-priority interrupt is received, the clock is restarted and the execution resumes from the next instruction.

It is also possible to put the sensor to sleep from the ISPU. This can only be done when the sensor is configured in a burst mode with the ISPU as off trigger. The mode can be configured by writing the BURST_CFG_[3:0] field in register CTRL1 (0x10).

Table 2. Accelerometer mode selection

BURST_CFG_[3:0]	Mode	On trigger	Off trigger
0000	Continuous	-	-
0001	Burst	User from interface	ISPU
0010		User from interface	FIFO
0100		Sleep counter	User from interface
0101		Sleep counter	ISPU
0110		Sleep counter	FIFO
1000		External trigger	User from interface
1001		External trigger	ISPU
1010		External trigger	FIFO

The sensor sleep can be triggered by executing a word store to address 0xC0000 (not mapped to any memory or register). For example:

```
stw [0xC0000], %r0
```

After putting the sensor to sleep, the ISPU is allowed to run until the ISPU itself is put to sleep. The ISPU will be allowed to wake up when the next on trigger is received, waking up the sensor.

3 Memory mapping

Memory addresses are expressed as 19-bit values in the ISPU. The address space is subdivided as follows:

- 0x02000 - 0x0FFFF → Data RAM (56 KB)
 - 0x02000 - 0x03FFF (8 KB)
 - 0x04000 - 0x05FFF (8 KB)
 - 0x06000 - 0x07FFF (8 KB)
 - 0x08000 - 0x0FFFF (32 KB)
- 0x40000 - 0x47FFF → Code RAM (32 KB)
- 0x80000 - 0x800FF → ISPU system registers (256 byte)
- 0x80800 - 0x808FF → ISPU registers (256 byte)
- 0x80900 - 0x809FF → External registers (256 byte)

Note: The data RAM is actually composed of 3 RAMs of 8 KB each and 1 RAM of 32 KB. This distinction is important, as the 32 KB data RAM uses a 128-bit hardware buffer to read (or write) four words with a single RAM access, thus saving power when accessing sequential data. The same applies to the 32 KB code RAM. For this reason, it is recommended to use the 32 KB RAMs for sequential data accesses, while the 8 KB RAMs should be preferably used for nonsequential accesses.

4 ISPU registers

Part of the registers used to interact with the ISPU fall under the group named "ISPU registers".

In order to access these registers from the device interface, it is necessary to set the PAGE_SEL bit in the RAM_ACCESS (0x01) register to 1. Once it is no longer needed to access these registers, the PAGE_SEL bit should be set back to 0 in order to restore access to the sensor registers.

5 External registers

Through the external registers interface it is possible to access a number of sensor registers from within the ISPU. In addition to being readable from the ISPU, some of them are also writable, thus allowing to change the configuration of the sensor directly from the ISPU, without the intervention of the external microcontroller.

However, write access to these registers is mutually exclusive with access from the device interface (SPI or I3C). In order to gain write access, the ISPU must set bit `EN_WRITE_EXT_REG` to 1 in the `EXT_CTRL` (0x80900) register. As soon as write access is not needed anymore, bit `EN_WRITE_EXT_REG` should be written back to 0.

Depending on the final system, it may be necessary to implement a mechanism to notify the external microcontroller or agree on switches in write access in order to avoid situations where the external microcontroller is unexpectedly unable to write to the sensor registers. The external microcontroller can also autonomously check the status by reading bit `ISPU_CTRL_ACCESS` in register `DEVICE_STATUS` (0x21). If the bit has a value of 1, the ISPU has gained write access to the registers.

Note: *After writing `EN_WRITE_EXT_REG` to 1, there is a delay of one clock cycle before the ISPU can write to the registers. This means that there must be another instruction before any write to the external registers is attempted. For this purpose, it is recommended to insert a `nop` instruction as shown in the following example.*

```
*((volatile uint8_t *)EXT_CTRL) = 0x01; // enable write in external registers
asm("nop"); // wait 1 clock cycle
*((volatile uint8_t *)CTRL2) = 0x20; // change accelerometer full-scale to 100 g
*((volatile uint8_t *)EXT_CTRL) = 0x00; // disable write in external registers
```

6 Sensor data

The data generated by the sensor can be read from the corresponding internal registers only accessible from the ISPU:

- Accelerometer
 - X-axis: ISPU_ARAW_X_L (0x80880), ISPU_ARAW_X_M (0x80881), ISPU_ARAW_X_H (0x80882), and ISPU_ARAW_X_HH (0x80883). ISPU_ARAW_X_L contains the least significant byte, ISPU_ARAW_X_HH contains the most significant byte.
 - Y-axis: ISPU_ARAW_Y_L (0x80884), ISPU_ARAW_Y_M (0x80885), ISPU_ARAW_Y_H (0x80886), and ISPU_ARAW_Y_HH (0x80887). ISPU_ARAW_Y_L contains the least significant byte, ISPU_ARAW_Y_HH contains the most significant byte.
 - Z-axis: ISPU_ARAW_Z_L (0x80888), ISPU_ARAW_Z_M (0x80889), ISPU_ARAW_Z_H (0x8088A), and ISPU_ARAW_Z_HH (0x8088B). ISPU_ARAW_Z_L contains the least significant byte, ISPU_ARAW_Z_HH contains the most significant byte.
- Temperature: ISPU_TEMP_L (0x808A4) contains the least significant byte, ISPU_TEMP_H (0x808A5) contains the most significant byte.

The value of each axis of the accelerometer is expressed as a 20-bit word in two's complement. However, for convenience, a sign extension to 32 bits is applied when the value is written to the registers, allowing to interpret it, within the ISPU code, using the `int32_t` type available in the `stdint.h` header of the standard C library. The value represents the raw data generated by the sensor, so, if necessary, the sensitivity must be applied by following the same procedure used for the regular sensor output registers.

The temperature value is expressed as a 16-bit word in two's complement. Within the ISPU code, it should be interpreted using the `int16_t` type available in the `stdint.h` header of the standard C library. Alternatively, it is possible to read two more bytes (available in the registers immediately after the first two bytes) and interpret the four bytes as a 32-bit word in two's complement (`int32_t` type), as a sign extension to 32 bits is applied when the value is written to the registers. The value represents the raw data generated by the sensor, so, if necessary, the sensitivity must be applied following the same procedure used for the regular sensor output registers.

7 Configuration

Besides reading the sensor data, it might be useful to receive inputs from the external microcontroller, for example to communicate the values of some configuration parameters at runtime instead of having to set them at compile time. For this reason, sixteen 8-bit general-purpose registers are available:

- ISPU_DUMMY_CFG_1 (0x52 / 0x80954)
- ISPU_DUMMY_CFG_2 (0x53 / 0x80955)
- ...
- ISPU_DUMMY_CFG_16 (0x61 / 0x80963)

These registers are readable and writable from the device interface (that is, from the external microcontroller) and are readable from the ISPU. Any value can be encoded in the 16 bytes.

If the ISPU_DUMMY_CFG registers are used to set the initial configuration parameters for an algorithm, it is recommended to write these values before enabling the algorithm (explained in [Section 11.6: Algorithms](#)), so that they are available for the algorithm initialization code.

An additional register, ISPU_IF2S_FLAG (0x0C-0x0D / 0x8080C-0x8080D), provides bits that can be set from the device interface (by the external microcontroller) and cleared by the ISPU. This register, unlike the ISPU_DUMMY_CFG registers, is part of the ISPU register page.

The intended use of the ISPU_IF2S_FLAG register is to raise flags to the ISPU, which can check the register, detect the bits that are set to 1, perform specific actions accordingly, and then clear the bits to signal that the request received from the device interface was served.

In order to set a bit of the ISPU_IF2S_FLAG register, it is sufficient to write 1 to it from the device interface. In order to clear the bit, the ISPU must write again 1 to it. While this might be counterintuitive, writing 1 means requesting the bit to be cleared.

The following code provides an example of how to use the ISPU_IF2S_FLAG within the ISPU:

```
uint16_t flag = *((volatile uint16_t *)ISPU_IF2S_FLAG) & 0x01; // get bit 0 value
if (flag) {
    // serve request from external microcontroller
    *((volatile uint16_t *)ISPU_IF2S_FLAG) = 0x01; // clear bit 0
}
```

8 Time-related inputs

In addition to the registers described in [Section 6: Sensor data](#) and [Section 7: Configuration](#), there are three more inputs accessible from the ISPU:

- **INTERNAL_FREQ_FINE** (0x50 / 0x80947) provides the difference of the real ODR (and timestamp rate) with respect to the nominal one.
- **TIMESTAMP0** (0x30 / 0x80930), **TIMESTAMP1** (0x31 / 0x80931), **TIMESTAMP2** (0x32 / 0x80932), **TIMESTAMP3** (0x33 / 0x80933) and **TIMESTAMP4** (0x34 / 0x80934) provide the timestamp information.
- **SLEEP_CNT_TIME_L** (0x62 / 0x80948) and **SLEEP_CNT_TIME_H** (0x63 / 0x80949) provide the current value of the sleep counter.

The **INTERNAL_FREQ_FINE** value is expressed as a percentage, encoded in two's complement with a resolution of 0.0078125%. The real ODR can be obtained as follows:

```
float odr = NOMINAL_ODR * (1.0f + *((volatile int8_t *)INTERNAL_FREQ_FINE) * 0.000078125f);
```

This correction can be used to compute an accurate delta time value, which may be necessary, for example, when integrating sensor data over time.

The timestamp is expressed as a 32-bit word with a bit resolution of 25 μ s. This resolution is the nominal value and, if needed, it can also be corrected using the **INTERNAL_FREQ_FINE** value as follows:

```
float timestamp_t = 25.0f / (1.0f + *((volatile int8_t *)INTERNAL_FREQ_FINE) * 0.000078125f);
```

Note: The timestamp generation must be enabled by setting the **TIMESTAMP_EN** bit to 1 in the **ISPU_CTRL2** (0x19 / 0x80919) register.

Note: The value contained in the **INTERNAL_FREQ_FINE** register is fixed and does not change at runtime. For this reason, the corrections may be applied only once, for example during the initialization routine of an algorithm.

Note: The **INTERNAL_FREQ_FINE** register can only be used if the **TPF** (0x20 / 0x80920) register contains a value different from zero.

The sleep counter value is expressed as a 16-bit word with a resolution dependent on the tick time selected with the **TICK_SEL** bit in the **SLEEP_CNT_CFG** (0x38 / 0x80917) register.

Table 3. Sleep counter resolution

TICK_SEL	Resolution [s]
0	13.1072
1	0.1024

The sleep counter may be used as an alternative source to track time. However, instead of manually checking its value, its main intended usage is to configure it to automatically trigger accelerometer data acquisition (using the sensor burst modes) or send an interrupt to the ISPU when a time threshold is reached (see [Section 11.5: Low-priority interrupts](#)).

9 Outputs

In order to export the results of the processing, the ISPU can write data to be read from the device interface to a set of general-purpose registers. In particular, sixteen 16-bit registers are available, for a total of 32 bytes that can be used to encode any information. These registers are part of the ISPU register page:

- ISPU_DOUT_00_L (0x10 / 0x80810), ISPU_DOUT_00_H (0x11 / 0x80811)
- ISPU_DOUT_01_L (0x12 / 0x80812), ISPU_DOUT_01_H (0x13 / 0x80813)
- ...
- ISPU_DOUT_15_L (0x2E / 0x8082E), ISPU_DOUT_15_H (0x2F / 0x8082F)

These registers are readable and writable from the ISPU, but are only readable by the external microcontroller from the device interface.

A block data update (BDU) mechanism in these registers is also available. Usually, if the output values are read synchronously with an interrupt signal generated by the ISPU when the outputs have just been written, this mechanism is not needed. However, if the output values are read asynchronously or if reading from the external microcontroller is slow, it might happen that some bytes of a particular output value are read before being updated with a new value by the ISPU, while the other bytes are read after. The resulting read value is thus corrupted. The BDU prevents this problem by blocking the refresh of all the bytes of the value at the start of the read until all bytes have been read. Note that the write of the new value by the ISPU is successful, but the old value is sent over the device interface until all bytes have been read.

The block data update can be enabled by setting to 1 both the BDU bit in the CTRL3 (0x12) register and the ISPU_BDU bit in the ISPU_CTRL1 (0x18) register.

Note: The BDU considers the 32 available bytes to be 8 values of 4 bytes each. If any output is written with some bytes in one 4-bytes block and some bytes in the next 4-bytes block, the BDU will not be able to ensure proper asynchronous reading.

10 FIFO

The ISPU can interact with the device FIFO in several ways:

- If the FIFO is configured in one of the modes that can change based on a trigger (continuous-to-FIFO, bypass-to-continuous, bypass-to-FIFO) and if FIFO_TRIGGER_CFG in register FIFO_CTRL2 (0x06) is set to 0, the ISPU can trigger a mode change (according to the FIFO configuration) by writing 1 in the ISPU2FIFO_INTERRUPT bit of register ISPU2FIFO_INT (0x8097F). In order to trigger again a mode change, the bit must be set back to 0 and then again to 1.
- The ISPU can write in the FIFO custom entries containing data to be read by the external microcontroller over the SPI or I3C device interface. Refer to [Section 10.1: FIFO write](#) for details.
- If bit FIFO_READ_FROM_ISPU in register FIFO_CTRL3 (0x07) is set to 1, the ISPU can read data from the FIFO and can configure the FIFO to change its behavior. When the ISPU has control of the FIFO, it is no longer accessible by the external microcontroller over the SPI or I3C device interface. Refer to [Section 10.2: FIFO read](#) for details.

10.1 FIFO write

In order to write an entry to the FIFO from the ISPU, these steps must be followed:

1. Write the FIFO tag (1 byte) in register ISPU_DOUT_11_L (0x80826). The tag value can be defined by the user, but it is recommended to use a different value than those already used by the device for accelerometer data, temperature data, and the timestamp.
2. Write the FIFO data (max 9 bytes) starting from register ISPU_DOUT_11_H (0x80827) until ISPU_DOUT_15_H (0x8082F).
3. Set register ISPU2FIFO_FLAG (0x8097C) to 0x01 to request writing the entry to the FIFO.
4. Wait for the write completion by waiting for register ISPU2FIFO_FLAG to go back to 0x00.

The following code shows how to implement the above procedure.

```
*((volatile uint8_t *)ISPU_DOUT_11_L) = 0x01; // write the user defined tag of the FIFO entry

// write the 72 bits of data of the FIFO entry
*((volatile uint8_t *)ISPU_DOUT_11_H) = 0x01;
*((volatile uint8_t *)ISPU_DOUT_12_L) = 0x02;
*((volatile uint8_t *)ISPU_DOUT_12_H) = 0x03;
*((volatile uint8_t *)ISPU_DOUT_13_L) = 0x04;
*((volatile uint8_t *)ISPU_DOUT_13_H) = 0x05;
*((volatile uint8_t *)ISPU_DOUT_14_L) = 0x06;
*((volatile uint8_t *)ISPU_DOUT_14_H) = 0x07;
*((volatile uint8_t *)ISPU_DOUT_15_L) = 0x08;
*((volatile uint8_t *)ISPU_DOUT_15_H) = 0x09;

*((volatile uint8_t *)ISPU2FIFO_FLAG) = 0x01; // FIFO write request
while (*((volatile uint8_t *)ISPU2FIFO_FLAG) != 0x01) // wait for request completion
;
```

10.2 FIFO read

In order to read the FIFO from the ISPU, first of all the FIFO_READ_FROM_ISPU bit in register FIFO_CTRL3 (0x07) must be set to 1.

When the FIFO_READ_FROM_ISPU bit is set to 1, the ISPU has full control of the FIFO, which is no longer governed by the configuration set in the FIFO_CTRL1 (0x05 / 0x80908), FIFO_CTRL2 (0x06 / 0x80909), and FIFO_CTRL3 (0x07 / 0x8090A) registers, but by the configuration set in the FIFO_CTRL1_ISPU (0x80910), FIFO_CTRL2_ISPU (0x80911), and FIFO_CTRL3_ISPU (0x80912) registers, which are totally independent from the other ones. This means that the configuration is not shared and must be set, if needed, twice, once from the interface, for when the FIFO is controlled and read from the interface, and once from the ISPU, for when the FIFO is controlled and read from the ISPU. The two configurations can be different.

Once the ISPU has control of the FIFO, these steps must be followed in order to read data from it:

1. Enable reading from the FIFO by setting bit FIFO2ISPU_READ_EN in register FIFO2ISPU_CTRL (0x8097E) to 1. This is necessary as the FIFO data is read from the same registers used to normally read the latest accelerometer data sample.
2. Set bit FIFO2ISPU_READ_REQ in register FIFO2ISPU_CTRL to 1 to request the read of one FIFO entry.

3. Wait for bit FIFO2ISPU_READ_VALID in register FIFO2ISPU_CTRL to be set to 1. When the bit transitions to 1, the request is completed and the data is ready to be read.
4. Read the FIFO2ISPU_TAG register (0x8097D), which contains the tag value of the FIFO entry just read. Process only entries with the tag 0x10, corresponding to accelerometer data entries.

Note: *If enabled, entries for temperature data and timestamp appear, but it is not possible to access their data.*

5. Read the FIFO entry data from the ISPU_ARAW registers to retrieve the accelerometer values contained in the FIFO entry. The registers can be read as described in [Section 6: Sensor data](#).
6. Set bit FIFO2ISPU_READ_REQ in register FIFO2ISPU_CTRL to 0.
7. Wait for bit FIFO2ISPU_READ_VALID in register FIFO2ISPU_CTRL to go back to 0.
8. Repeat from step 2 to read more entries. When register FIFO2ISPU_TAG has a value of 0x00, there are no more entries to read.
9. Disable reading from the FIFO by setting bit FIFO2ISPU_READ_EN in register FIFO2ISPU_CTRL to 0.

The following code shows how to implement the above procedure to read all the entries available in the FIFO and compute the mean value of each accelerometer axis.

```
int32_t acc_mean[3] = { 0, 0, 0 }, acc_n = 0;

*((volatile uint8_t *)FIFO2ISPU_CTRL) = 0x04; // enable FIFO read

uint8_t tag = 0;
do {
    *((volatile uint8_t *)FIFO2ISPU_CTRL) = 0x06; // request entry read
    while (((volatile uint8_t *)FIFO2ISPU_CTRL) & 0x01) == 0); // wait for entry read

    tag = *((volatile uint8_t *)FIFO2ISPU_TAG);

    if (tag == 0x10) { // read and process accelerometer data from FIFO entry
        acc_mean[0] += *((volatile int32_t *)ISPU_ARAW_X);
        acc_mean[1] += *((volatile int32_t *)ISPU_ARAW_Y);
        acc_mean[2] += *((volatile int32_t *)ISPU_ARAW_Z);

        acc_n++;
    }

    *((volatile uint8_t *)FIFO2ISPU_CTRL) = 0x04; // clear request
    while (((volatile uint8_t *)FIFO2ISPU_CTRL) & 0x01) != 0);
} while (tag); // loop until FIFO is empty

*((volatile uint8_t *)FIFO2ISPU_CTRL) = 0x00; // disable FIFO read

acc_mean[0] /= acc_n;
acc_mean[1] /= acc_n;
acc_mean[2] /= acc_n;
```

Note: *If reading the FIFO immediately after the ISPU is woken up by the generation of a new accelerometer sample, it is possible that the same sample is not yet available in the FIFO. It is recommended to use the FIFO low-priority interrupt to trigger the read when the FIFO contains the desired number of samples (see [Section 11.5: Low-priority interrupts](#)).*

11 Inbound interrupts

The ISPU can receive a number of interrupt requests (IRQ), as shown in [Table 4](#). These should not be confused with the interrupts generated by the device on the INT1 and INT2 pins, described in [Section 12: Outbound interrupts](#).

Each interrupt request corresponds to a hardware signal that can be sent to the ISPU to execute a specific routine upon request. The user must create an interrupt vector table (IVT) to map each interrupt request to a proper routine to serve it. Each entry of the IVT must be implemented as a jump to the routine. The jump instruction must be placed at the address specified in [Table 4](#). As evident from the addresses, the IVT must be placed at the beginning of the 32 KB code RAM, starting from address 0x40000.

Note: An interrupt with a higher priority level can interrupt one with a lower priority level.

Note: When using multiple interrupts with the same level of priority, it must be ensured that during the execution of an interrupt routine another interrupt with a higher position in the interrupt table (but same priority level) is not generated. For example, it is not allowed to generate the interrupt for algorithm 01 from the interrupt routine of algorithm 00.

Table 4. ISPU interrupt requests

Program address	Interrupt definition	Priority level
0x40000	Trap	4
0x40004	Low-priority interrupt	0
0x40010	Algorithm 00 run	0
0x40014	Algorithm 01 run	0
0x40018	Algorithm 02 run	0
0x4001C	Algorithm 03 run	0
0x40020	Algorithm 04 run	1
0x40024	Algorithm 05 run	1
0x40028	Algorithm 06 run	1
0x4002C	Algorithm 07 run	1
0x40030	Algorithm 08 run	1
0x40034	Algorithm 09 run	1
0x40038	Algorithm 10 run	1
0x4003C	Algorithm 11 run	1
0x40040	Algorithm 12 run	2
0x40044	Algorithm 13 run	2
0x40048	Algorithm 14 run	2
0x4004C	Algorithm 15 run	2
0x40050	Overflow	2
0x40078	Data valid / high-priority interrupt	3
0x40080	Boot	n/a
0x40090	Algorithm 00 initialization	0
0x40094	Algorithm 01 initialization	0
0x40098	Algorithm 02 initialization	0
0x4009C	Algorithm 03 initialization	0
0x400A0	Algorithm 04 initialization	1
0x400A4	Algorithm 05 initialization	1

Program address	Interrupt definition	Priority level
0x400A8	Algorithm 06 initialization	1
0x400AC	Algorithm 07 initialization	1
0x400B0	Algorithm 08 initialization	1
0x400B4	Algorithm 09 initialization	1
0x400B8	Algorithm 10 initialization	1
0x400BC	Algorithm 11 initialization	1
0x400C0	Algorithm 12 initialization	2
0x400C4	Algorithm 13 initialization	2
0x400C8	Algorithm 14 initialization	2
0x400CC	Algorithm 15 initialization	2

11.1 Boot

The interrupt request at 0x40080 is generated when booting the ISPU (see [Section 16: Boot procedure](#)) and must jump to a procedure that executes the startup code and then, normally, puts the ISPU in the sleep state to wait to be woken up for data processing.

11.2 Traps

An interrupt is generated at address 0x40000 in case of a trap. There are four types of trap exception that the ISPU can generate:

- **INSTR_BUS_ERROR**: the instructions memory controller signaled a bus error.
- **DATA_BUS_ERROR**: the data memory controller signaled a bus error.
- **ILLEGAL_INSTRUCTION**: the fetched instruction could not be decoded into a legal instruction.
- **MISALIGNED_ACCESS**: an access to a misaligned address was attempted. This trap can be generated, for example, attempting a load of a 32-bit word from an address not aligned to 32 bits with a single instruction: `ldw %r0, [0x02001]`. The same happens with 16-bit words not aligned to 16 bits and with store instructions.

This is the interrupt with the highest level of priority, meaning that it can interrupt any other interrupt service routine.

11.3 Data valid / high-priority interrupt

If the accelerometer rate is not configured in idle and the ISPU rate is configured to a value different from zero, a signal is generated when new valid accelerometer data is available from the sensor, at an interval corresponding to the ISPU rate. When such signal is received, the ISPU is woken up if previously asleep.

In addition, upon receiving the signal, an interrupt can be generated at address 0x40078. The generation of this interrupt is enabled if the **EN_DVAL_INT** bit in **ISPU_GLB_CALL_EN** (0x80800) is set to 1.

This interrupt is generated also if the ISPU is already awake, and it is high-priority, meaning that it can interrupt most other interrupt service routines (the trap interrupt being the only exception). Most importantly, it can interrupt the execution of algorithms, thus allowing the user to store the new data for later processing, and then resume the algorithms execution.

11.4 Overrun

The overrun interrupt is generated at address 0x40050 if the ISPU is still executing when a new accelerometer data sample is generated from the sensor. Bit **EN_ORUN_INT** in register **ISPU_GLB_CALL_EN** (0x80800) must be set to 1 to enable the generation of the interrupt.

This interrupt can be useful to perform specific actions on the overrun event. However, if the goal is to retrieve the new data sample and store it for later processing, it is recommended to use the data valid interrupt, which is always triggered, even if an overrun does not occur (see [Section 11.3](#)).

It is also possible to monitor if an overrun has occurred by checking bit **ORUN** in register **ISPU_STATUS** (0x04) in the ISPU register page.

Note: *In order to ensure the proper functioning of the ISPU, the overrun interrupt must not be enabled if using any of the algorithms that have the same priority level of the overrun interrupt (from algorithm 12 to algorithm 15). See also the notes in [Section 11: Inbound interrupts](#).*

11.5 Low-priority interrupts

The low-priority interrupt is generated at address 0x40004. It can be triggered by various sources:

- **FIFO:** the interrupt is triggered when the entries stored in the FIFO reach the configured watermark threshold or when the FIFO full event is generated (FIFO will be full at the next ODR). In order to enable this interrupt source, bit LOPRIO_FIFO_EN in register ISPU_LOPRIO_EN (0x39) must be set to 1.
- **Sleep counter:** the interrupt is triggered when the sleep counter reaches the configured threshold. In order to enable this interrupt source, bit LOPRIO_SLEEP_CNT_EN in register ISPU_LOPRIO_EN must be set to 1.
- **External trigger:** the interrupt is triggered on the rising edge of the signal on the INT2 pin. It is possible to instead trigger it on the falling edge by setting bit INT2_IN_LH in the CTRL4 (0x13) register to 1. In order to enable this interrupt source, bit LOPRIO_EXT_TRG_EN in register ISPU_LOPRIO_EN must be set to 1.
- **User trigger:** the interrupt is triggered when the user changes the value of bit LOPRIO_USER_TRG in register ISPU_CTRL1 (0x18) from 0 to 1. In order to enable this interrupt source, bit LOPRIO_USER_TRG_EN in register ISPU_LOPRIO_EN must be set to 1.
- **Temperature:** the interrupt is triggered when a new sample of temperature data is generated by the sensor. The rate of the temperature data is 52 Hz. In order to enable this interrupt source, bit LOPRIO_TEMP_EN in register ISPU_LOPRIO_EN must be set to 1.

In addition to the specific bits for each trigger source, the EN_LOPRI_INT bit in ISPU_GLB_CALL_EN (0x80800) must be set to 1.

The low-priority interrupt can wake up the ISPU if asleep, but it cannot interrupt other interrupt service routines, as it has the lowest level of priority.

Since multiple sources can trigger this interrupt, the register ISPU_LOPRIO_INT (0x80978) is provided to check which sources are responsible for the interrupt generation. Even if just a single source has been configured, it is mandatory to read this register within the interrupt service routine. The read of the register causes the clearing of all the bits in it. If the register is not read, and thus cleared, the interrupt will be immediately re-triggered when returning from the interrupt service routine.

Note: *It is not allowed to generate an interrupt for algorithms with the same level of priority of the low-priority interrupt (from algorithm 00 to algorithm 03) from its interrupt routine. See also the notes in [Section 11: Inbound interrupts](#).*

11.6 Algorithms

The ISPU is an embedded core that can be freely programmed to handle the processing of data as desired. However, the device also implements some mechanisms at a hardware level to facilitate the execution of multiple algorithms or processing logics and the management of their lifecycle.

As indicated in [Table 4](#), there is hardware support for up to 16 algorithms, from algorithm 00 to algorithm 15. The device is able to generate, for each algorithm, an interrupt request to execute the algorithm initialization code and an interrupt request to execute the run code of the algorithm.

The generation of the interrupt requests related to the algorithms is controlled by the user by writing, from the ISPU code, to the appropriate registers. First of all, the EN_ALGO_INT bit in ISPU_GLB_CALL_EN (0x80800) must be set to 1 in order to enable the interrupt request generation for all algorithms.

Then, in order to generate an interrupt request for an algorithm, the corresponding bit must be set to 1 in registers ISPU_CALL_EN_1 (0x808B8), ISPU_CALL_EN_2 (0x808B9), ISPU_CALL_EN_3 (0x808BA) and ISPU_CALL_EN_4 (0x808BB).

Note: *The 16 algorithm bits start from bit 4 of the ISPU_CALL_EN_1 register, so, for example, in order to generate an interrupt request for algorithm 04, bit ISPU_CALL_EN_ALGO4, which is bit 0 of the ISPU_CALL_EN_2 register, must be set to 1.*

When this bit is set to 1, the interrupt request for the corresponding algorithm is generated and the bit value remains at 1 until the interrupt service routine of the algorithm has completed. Once the routine has completed, the bit is automatically reset to 0. This allows monitoring the status of the execution of the algorithms and waiting for their completion before putting the ISPU in the sleep state to wait for new data to process.

When an interrupt is requested for an algorithm, the interrupt request generated normally corresponds to the routine for running the algorithm, except for the first time, when an interrupt request is generated to execute the algorithm initialization routine. In order to trigger again the execution of the algorithm initialization code, it is necessary to disable and re-enable the algorithm, as explained in the following paragraphs.

Of course the interrupt requests for the algorithms can be generated at any time, but it is recommended to generate them after the ISPU has been woken up due to new data being available for processing. In this way the algorithms can be executed to process the new data and then the ISPU can be again put in the sleep state.

Figure 2 in [Section 12: Outbound interrupts](#) shows the usage of the ISPU_CALL_EN registers in the recommended processing cycle.

If the execution of multiple algorithms is requested at the same time, the algorithm routines are executed in reverse order, from algorithm 15 to algorithm 00.

Note: *During the execution of an algorithm routine, it is not allowed to generate the interrupt for another algorithm with a higher position in the interrupt table (but same priority level). For example, it is not allowed to generate the interrupt for algorithm 01 from the interrupt routine of algorithm 00. See also the notes in [Section 11: Inbound interrupts](#).*

Note: *It is not allowed to generate an interrupt for algorithms from 00 to 03 from the interrupt routine of the low-priority interrupt ([Section 11.5: Low-priority interrupts](#)). See also the notes in [Section 11: Inbound interrupts](#).*

A set of registers, part of the ISPU register page, is available to enable or disable each algorithm from the device interface:

- ISPU_ALGO1 (0x70 / 0x80870) allows enabling / disabling algorithms 00 to 07.
- ISPU_ALGO2 (0x71 / 0x80871) allows enabling / disabling algorithms 08 to 15.

As mentioned above, disabling and re-enabling an algorithm using these registers resets an internal flag that causes the algorithm initialization routine to be called the next time an interrupt request is generated. Aside from this, it is up to the user to use these registers within the ISPU code to enable and disable the execution of the algorithms, which exclusively depends on the value of the EN_ALGO_INT bit and on which bits are written to 1 in the ISPU_CALL_EN registers. For example, when the ISPU is woken up, the ISPU_ALGO registers can be read in order to determine which algorithms are enabled and set the corresponding bits in the ISPU_CALL_EN registers, thus executing only the enabled algorithms. This can be achieved as follows:

```
*((volatile uint32_t *)ISPU_CALL_EN) = *((volatile uint16_t *)ISPU_ALGO) << 4;
```

The bits in the ISPU_ALGO registers can also be cleared from the ISPU in order to disable the algorithms. This can be achieved by writing 1 to the bit corresponding to the algorithm to disable. While this might be counterintuitive, writing 1 means requesting the bit to be cleared. However, it is not possible to enable the algorithms from the ISPU.

12 Outbound interrupts

In order to keep the external microcontroller sleeping most of the time, it is necessary to avoid polling for new results from the ISPU. For this reason, it is possible to generate interrupt signals on the INT1 and INT2 pins of the device to wake up the external microcontroller only when necessary.

First of all, the interrupts routing to the pins must be enabled:

- Setting INT1_ISPU to 1 in the INT_CTRL4 (0x6E) register enables routing the interrupts to the INT1 pin.
- Setting INT2_ISPU to 1 in the INT_CTRL4 (0x6E) register enables routing the interrupts to the INT2 pin.

The interrupts can then be configured in three different modes by setting the LATCHED_[1:0] field in the ISPU_CONFIG (0x02) register, which is part of the ISPU register page.

Table 5. ISPU interrupt modes

LATCHED_[1:0]	Mode
00	Both the interrupt status and pins are pulsed
01	The interrupt status is latched while the interrupt pins are pulsed
1x	Both the interrupt status and pins are latched

If the interrupt pins are configured in pulsed mode, they must be directly controlled from the ISPU code by writing 0 or 1 to the INT1 and INT2 bits in register ISPU_INT_PIN (0x5C / 0x8085C). Writing 0 causes the pin to be set to the inactive level, writing 1 causes the pin to be set to the active level.

Note that the active level of the interrupt pins can be either set to high or low depending on the value of the INT_ACTIVE_LEVEL bit in the IF_CFG (0x03) register. If the INT_ACTIVE_LEVEL bit is set to 0, the interrupt pins are active high, otherwise they are active low.

In order to generate a pulsed interrupt signal, the INT1 or INT2 bit (based on the desired pin) can be set to 1 at the end of the processing and before going to sleep (if an interrupt must be raised) and set back to 0 when the ISPU is woken up the next time (usually when the next data sample is generated by the sensor). While this is the most common usage, it must be noted that it is entirely up to the user to decide when to set the interrupt pin to the active level and when to reset it to the inactive level. In any case, the pin status is entirely managed by the ISPU code. The external microcontroller does not need to perform any action to reset the interrupt line.

If the interrupts pins are instead configured in latched mode, the ISPU is unable to directly set the pin level. Instead, the interrupt line is automatically controlled by the device.

A latched interrupt is generated when one or multiple interrupt flags are raised in the interrupt status registers, which are part of the ISPU register page:

- ISPU_INT_STATUS1 (0x58 / 0x80858) contains the interrupt flags for algorithms 00 to 07.
- ISPU_INT_STATUS2 (0x59 / 0x80859) contains the interrupt flags for algorithms 08 to 15.

In addition, the bits in the ISPU_INT1_CTRL or ISPU_INT2_CTRL registers corresponding to the raised interrupt flags must be set to 1, otherwise the flags are ignored and the latched interrupt is not generated on the pins. These control registers, part of the ISPU register page, must be written from the device interface and can be used to enable or disable the routing of the interrupts to the INT1 and INT2 pins for each algorithm:

- ISPU_INT1_CTRL1 (0x50 / 0x80850) allows enabling / disabling routing to INT1 for algorithms 00 to 07.
- ISPU_INT1_CTRL2 (0x51 / 0x80851) allows enabling / disabling routing to INT1 for algorithms 08 to 15.
- ISPU_INT2_CTRL1 (0x54 / 0x80854) allows enabling / disabling routing to INT2 for algorithms 00 to 07.
- ISPU_INT2_CTRL2 (0x55 / 0x80855) allows enabling / disabling routing to INT2 for algorithms 08 to 15.

When the pins are configured in latched mode, the interrupt line is set back to the inactive level when the external microcontroller reads the raised bits in the interrupt status registers.

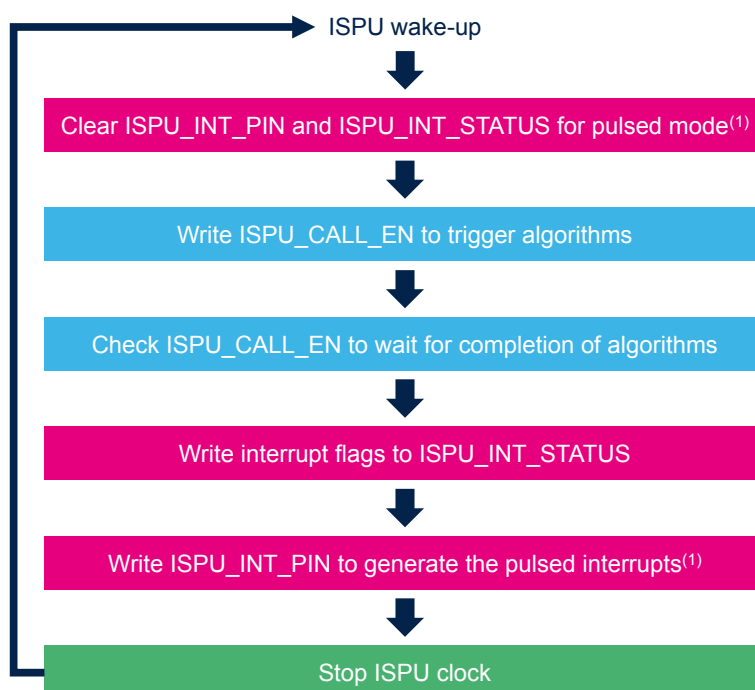
Note that the interrupt flags in the interrupt status registers are not automatically generated, but it is up to the user to write, from the ISPU code, the corresponding bit to 1 when an algorithm has to trigger an interrupt (when the pins are configured in pulsed mode, this should be done before generating the actual interrupt signal on the pin).

If the interrupt status is configured in latched mode (independently from the configuration of the pins), then the bits are automatically reset to 0 when reading the interrupt status registers from the device interface. If the interrupt status is configured in pulsed mode, it is once again up to the user to set the bits back to 0. The reset of these bits should be performed at the same time as the `ISPU_INT_PIN` register is written in order to set the interrupt pin back to the inactive level (when the pin is also set in pulsed mode), for example, when the ISPU is woken up because the next data sample is available.

The purpose of the interrupt status registers is to allow the external microcontroller to understand which algorithms have generated the interrupt.

Figure 2 shows the usage of the `ISPU_INT_PIN` and `ISPU_INT_STATUS` registers in the recommended processing cycle. Note that, as explained above, writing the `ISPU_INT_PIN` register does not have an effect if the pins are configured in latched mode, but should be done anyway to support pulsed mode with the same code.

Figure 2. ISPU recommended processing cycle



1. Step not necessary in latched mode.

A copy of the interrupt flags is also available in registers `ISPU_INT_STATUS1_MAINPAGE` (0x1A) and `ISPU_INT_STATUS2_MAINPAGE` (0x1B), which are readable without having to enable access to the ISPU register page. If the interrupt status is configured in latched mode, reading these registers is equivalent to reading the `ISPU_INT_STATUS` registers for clearing the status bits and the interrupt signal on the pins.

Of course the bits of the interrupt status registers can be used freely, so if the concept of multiple algorithms is not used, the bits can be used, for example, to distinguish different types of interrupts related to the same algorithm.

Note that, if the interrupt pins are configured in pulsed mode, the `ISPU_INT1_CTRL` and `ISPU_INT2_CTRL` registers produce no effect on the device, and it is up to the user to use them within the ISPU code to generate an interrupt on the pins only if the bit corresponding to the algorithm that generates the interrupt is set to 1, meaning that the interrupt must be routed to the pin. A generic interrupt can be generated at software level and then, based on the `ISPU_INT1_CTRL` and `ISPU_INT2_CTRL` register values, it can be decided whether to generate an interrupt on the desired interrupt pin. This can be achieved with some simple masking, as shown in the following code.

```

uint8_t int_pin = 0u;
int_pin |= ((int_status & *((volatile uint16_t *)ISPU_INT1_CTRL)) > 0u) ? 0x01u : 0x00u;
int_pin |= ((int_status & *((volatile uint16_t *)ISPU_INT2_CTRL)) > 0u) ? 0x02u : 0x00u;

*((volatile uint16_t *)ISPU_INT_STATUS) = int_status;
*((volatile uint8_t *)ISPU_INT_PIN) = int_pin;
  
```

In the snippet of code above, the `int_status` variable contains the interrupt flags for all algorithms. The value of this variable is masked with the content of the interrupt control registers to determine whether to actually trigger an interrupt on each of the two pins (INT1 and INT2).

In addition to the physical interrupts, an additional register, `ISPU_S2IF_FLAG` (0x0Eh-0xFh / 0x8080E-0x8080F), provides bits that can be set by the ISPU and cleared from the device interface by the external microcontroller. This register is part of the ISPU register page.

The intended use of the `ISPU_S2IF_FLAG` register is to raise flags to the external microcontroller, which can check the register, detect the bits that are set to 1, perform specific actions accordingly, and then clear the bits to signal that the request received from the ISPU was served. In this sense, these bits can be used to implement a sort of software interrupt. Of course, in this case, the external microcontroller needs to poll the register to detect when the flags are raised. Alternatively, this register could be used in combination with a physical interrupt to ask the external microcontroller to perform some actions when the interrupt is received.

In order to set a bit of the `ISPU_S2IF_FLAG` register, it is sufficient to write 1 to it from the ISPU. In order to clear the bit, the bit must be written again to 1 from the device interface. While this might be counterintuitive, writing 1 means requesting the bit to be cleared.

13 Hardware accelerators

The ISPU provides a set of hardware blocks, usable from the system registers, to accelerate loops, FFT computation, circular buffers, and square root computation. This section provides details on what each hardware block does and how to use it.

In addition to the hardware blocks usable from the system registers, a specific assembly instruction for performing accumulate and multiply operations is described, as its usage is essential to accelerate some types of calculations.

13.1 Zero-overhead loop

The zero-overhead loop (ZOL) accelerator allows implementing code loops without their inherent overhead. Using the accelerator, there is no need to explicitly track the number of iterations already performed, compare it with the total number of iterations to perform and jump back to the beginning of the loop when there are more iterations to perform.

The accelerator can be used by properly setting the following ISPU system registers:

- **ISPU_LOOP_START** (0x80080). Here the user must write the address of the first instruction of the loop. The address must be aligned to 2 bytes (as instructions always are on the ISPU).
- **ISPU_LOOP_END** (0x80084). Here the user must write the address of the last instruction of the loop. The address must be aligned to 2 bytes (as instructions always are on the ISPU).
- **ISPU_LOOP_CNT** (0x80088). Here the user must write, in the 10 least-significant bits (**LOOP_CNT[9:0]**), the number of iterations to perform minus 1 (0 means 1 iteration). The maximum number of iterations is 1024. Bit 10 (**LOOP_EN**) must be set to 1 to enable the accelerator.

Before the loop instructions, the user must set the loop start address in **ISPU_LOOP_START**, the loop end address in **ISPU_LOOP_END**, the number of iterations in **LOOP_CNT[9:0]**, and enable the accelerator by setting the **LOOP_EN** bit to 1.

The following example shows how to configure a loop of 20 iterations to accumulate 2 and 5 in, respectively, **r0** and **r1** registers. At the end of the loop **r0** will contain the value 40 (2 times 20 iterations) and **r1** will contain 100 (5 times 20 iterations).

```
; ZOL configuration
movp %r0, loop_start      ; set loop start address
stp [ISPU_LOOP_START], %r0
movp %r0, loop_end        ; set loop end address
stp [ISPU_LOOP_END], %r0
movh %r0, 19              ; 20 iterations (20 - 1 = 19)
bset %r0, 10              ; set bit 10 to 1 (LOOP_EN)
sth [ISPU_LOOP_CNT], %r0

; initialize values used in the loop
movw %r0, 0               ; initialize accumulator 1 to 0
movw %r1, 0               ; initialize accumulator 2 to 0

; loop
loop_start:
    addw %r0, 2            ; add 2 to accumulator 1
loop_end:
    addw %r1, 5            ; add 5 to accumulator 2
```

As shown in the example, it is not necessary to manually compute the start and end addresses. Labels (**loop_start** and **loop_end** in the example) can be used to mark the start and end instructions and then to reference the corresponding addresses. In this example, the loop is composed of the two **addw** instructions, which are executed 20 times.

13.2 FFT (Fast Fourier Transform)

The FFT accelerator allows implementing a radix-2 Cooley-Tukey FFT algorithm, accelerating various operations by reducing the cost of data access in memory. Three different accelerators can be distinguished:

- Bit reverse addressing accelerator, allowing faster access to the data for performing the bit reversal step of the algorithm.
- Data accelerator, allowing faster access to the data buffer for reading input data and storing output data during the calculations.
- Twiddle table accelerator, allowing faster access to the twiddle factors to use in the computations.

In the bit reversal step of the algorithm, the data must be reordered in a manner where each element must be moved to an index in the buffer that has a value obtained by reversing the bits of the value of the original index. For example, in a buffer of 16 elements, the data at index 2 (0010) must be moved to index 4 (0100).

The bit reverse addressing accelerator can be used by means of the following ISPU system registers.

ISPU_FFT_N_ELEM_BRA (0x800E0). Here the user must write the number of points of the FFT according to Table 6.

Table 6. FFT_N_BRA[2:0] values

FFT_N_BRA[2:0]	Number of points
000	16
001	32
010	64
011	128
100	256
101	512
110	1024

This is necessary for the accelerator to know on how many bits to perform the bit reversal.

Note: Each point is a complex number. If a real FFT is computed by packing the values in the real and imaginary parts of the complex numbers, this register must be set to the number of complex points (half of the original real points).

ISPU_FFT_CURR_BRA (0x800DC). Here the address to the current element to be bit reversed is stored. At the beginning of the bit reversal procedure, the user must write the address of the first element of the buffer to be bit reversed (or of the second element, since the first element will never have to bit reversed) to this register. The address must be aligned to 4 bytes (bits 0 to 1 are fixed to zero).

ISPU_FFT_MATCH_BRA (0x800D8). Here the user can read the currently pointed element and the element at the bit-reversed index, and write the new values to perform the swap. For each element of the buffer, the following reads and writes using register ISPU_FFT_MATCH_BRA must be performed:

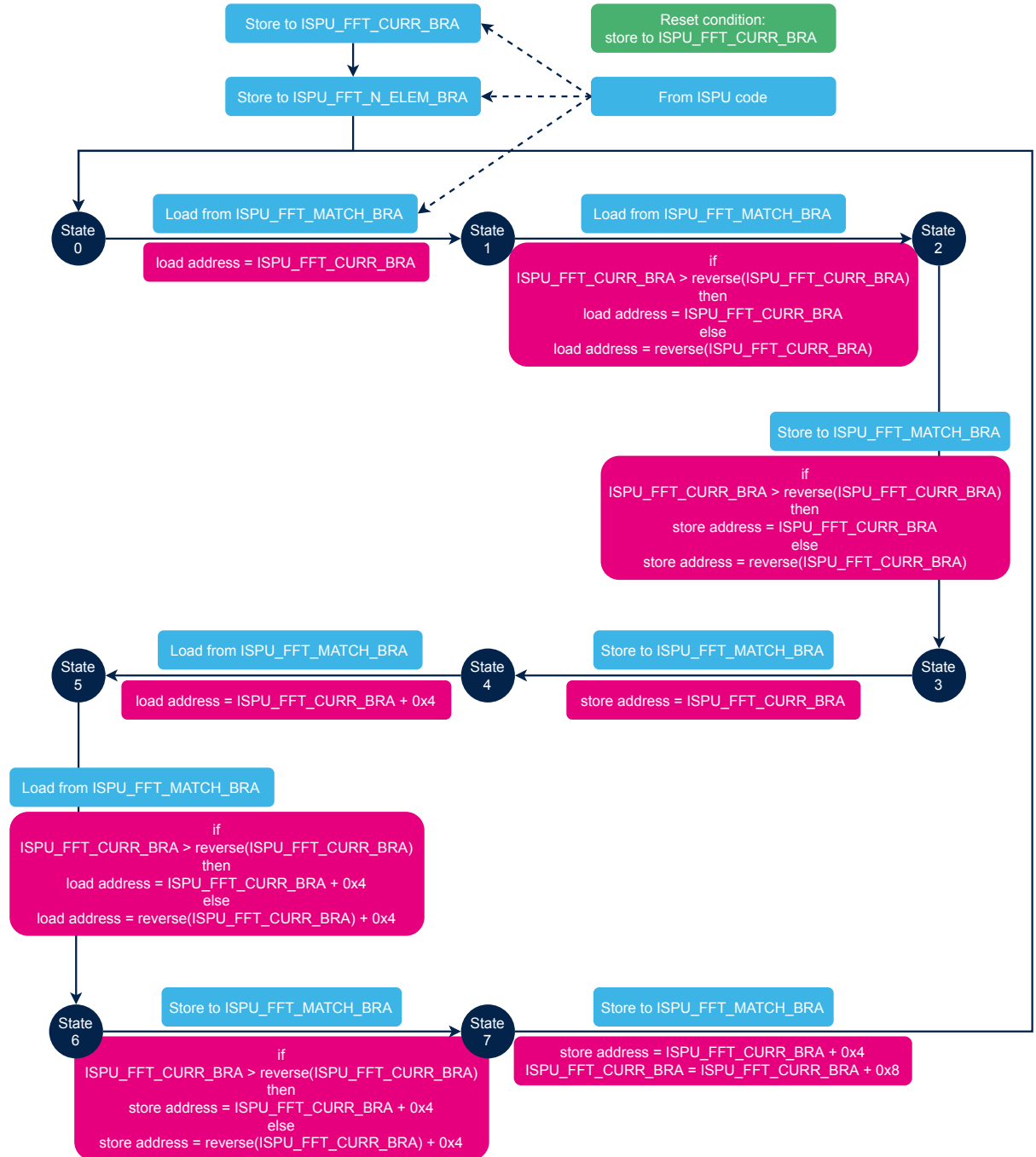
1. Read to get the real part of the current element.
2. Read to get the real part of the element at the bit-reversed index.
3. Write to set the real part of the element at the bit-reversed index to the value read at step 1.
4. Write to set the real part of the current element to the value read at step 2.
5. Read to get the imaginary part of the current element.
6. Read to get the imaginary part of the element at the bit-reversed index.
7. Write to set the imaginary part of the element at the bit-reversed index to the value read at step 5.
8. Write to set the imaginary part of the current element to the value read at step 6.

Note: At steps 2, 3, 6, and 7 the element at the bit-reversed index is accessed only if the index is lower or equal to the bit-reversed index, otherwise the current element is accessed. This is to avoid re-swapping the elements when the current element was already swapped in a previous iteration in which it was the element at the bit-reversed index.

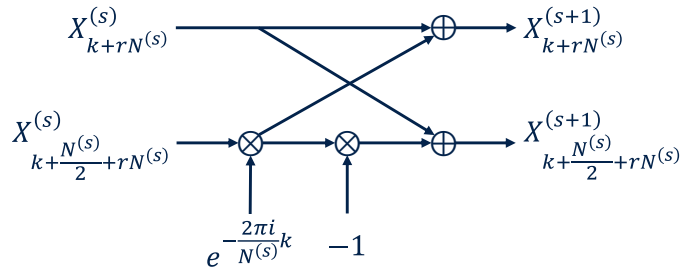
After performing the steps above, the address in ISPU_FFT_CURR_BRA is incremented by 8 to get ready to perform the bit reversal of the next element (a complex number in single-precision floating-point format is 8 bytes).

ISPU_FFT_STATE_BRA (0x800D4). Here the user can find the state the accelerator is currently in, as shown in Figure 3. It is not necessary to use this register to implement the bit reversal.

Figure 3. FFT bit reverse addressing accelerator FSM



The core of the FFT computation when implementing a radix-2 Cooley-Tukey algorithm is a series of butterfly operations. Figure 4 shows the generic expression of a single butterfly operation for a decimation-in-time (DIT) implementation.

Figure 4. FFT butterfly operation (DIT)


If the number of (complex) points of the FFT is N , the radix-2 Cooley-Tukey algorithm requires $\log_2(N)$ stages of computation. In each stage, $\frac{N}{N^{(s)}}$ sub-FFTs must be computed, where $N^{(s)}$ is the size of each sub-FFT and s is the stage being computed.

$$N^{(s)} = 2^s$$

$$s = 1, \dots, \log_2(N)$$

Referring to [Figure 4](#), r is the index of the sub-FFT currently being computed and k is the index of the element of the sub-FFT being computed.

$$r = 0, \dots, \frac{N}{N^{(s)}} - 1$$

$$k = 0, \dots, \frac{N^{(s)}}{2} - 1$$

As it can be observed in [Figure 4](#), the data X involved in the butterfly operations must be accessed in a specific, non-contiguous way. The data accelerator allows directly accessing all the data in the correct order without the need to manually compute the indexes. The accelerator can be used by means of the following ISPU system registers.

ISPU_FFT_TWIDDLE (0x800CC). Here, at the beginning of each stage, the user must write the number of twiddle factors needed for the specific stage: $\frac{N^{(s)}}{2} = 2^{(s-1)}$. The maximum is 512 (for the last stage in a DIT implementation with 1024 points).

This is necessary for the accelerator to know the size of each sub-FFT ($N^{(s)}$), how far are the two elements of the buffer involved in each butterfly operation ($\frac{N^{(s)}}{2}$) and how much to advance in the data buffer after each sub-FFT is completed ($\frac{N^{(s)}}{2}$).

When ISPU_FFT_TWIDDLE is written, an internal counter is initialized to the register value.

ISPU_FFT_CURR_D1 (0x800C4). Here the address to the first element ($X_{k+rN^{(s)}}$) for the current butterfly operation is stored. At the beginning of each stage, the user must write to this register the address of the first element of the data buffer. The address must be aligned to 4 bytes (bits 0 to 1 are fixed to zero). The address of the first element of the data buffer must be further aligned based on the number of points, as shown in [Table 7](#).

Table 7. FFT data buffer alignment

Number of points (N)	Alignment [bytes]
16	128
32	256
64	512
128	1024
256	2048
512	4096
1024	8092

When writing this register, ISPU_FFT_CURR_D2 is automatically set to the same value plus the number of twiddles multiplied by 8 (each element of the data buffer is a complex number, which is 8 bytes in single-precision floating point format). This is because the number of twiddles corresponds to the distance between the first and the second elements involved in the butterfly operation.

Note: *This register must be written after ISPU_FFT_TWIDDLE for the address in ISPU_FFT_CURR_D2 to be set correctly.*

ISPU_FFT_CURR_D2 (0x800C8). Here the address to the second element ($X_k + \frac{N^{(s)}}{2} + rN^{(s)}$) for the current butterfly operation is stored. Normally, the user does not need to explicitly write this register, as it is automatically set when writing ISPU_FFT_CURR_D1 (see description of the register above). The address must be aligned to 4 bytes (bits 0 to 1 are fixed to zero).

Bits after bit 12 are fixed to zero in this register. The high part of the address is taken from ISPU_FFT_CURR_D1.

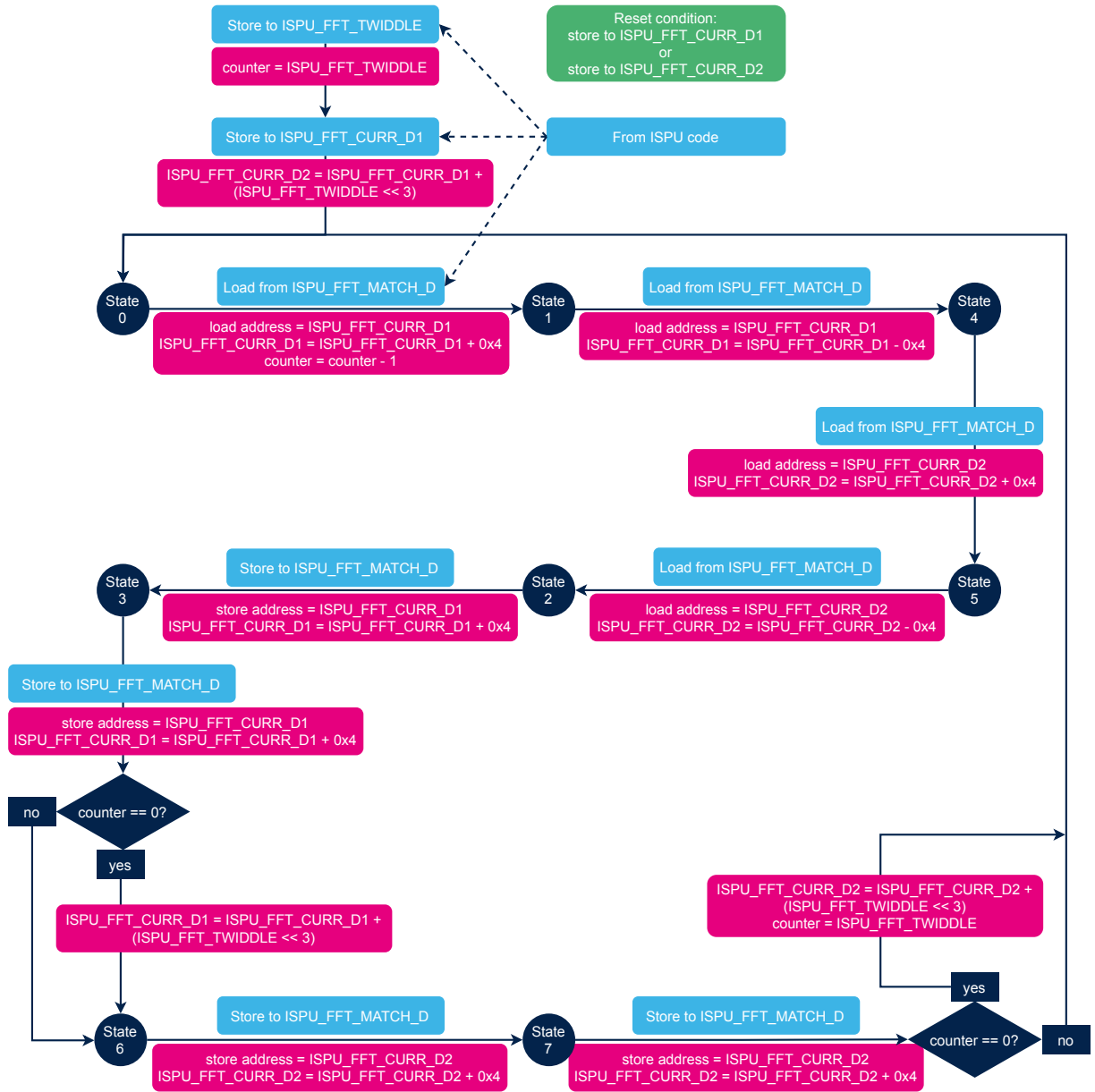
ISPU_FFT_MATCH_D (0x800C0). Here the user can read the currently pointed first ($X_k + rN^{(s)}$) and second ($X_k + \frac{N^{(s)}}{2} + rN^{(s)}$) elements for the butterfly operation, and write the results of the operation to the same positions in the buffer. For each butterfly operation, the following reads and writes using register ISPU_FFT_MATCH_D must be performed:

1. Read to get the real part of the first element.
2. Read to get the imaginary part of the first element.
3. Read to get the real part of the second element.
4. Read to get the imaginary part of the second element.
5. Write to set the real part of the result to store at the position of the first element.
6. Write to set the imaginary part of the result to store at the position of the first element.
7. Write to set the real part of the result to store at the position of the second element.
8. Write to set the imaginary part of the result to store at the position of the second element.

After performing the steps above, the internal counter is decremented by 1, and the address values of ISPU_FFT_CURR_D1 and ISPU_FFT_CURR_D2 are incremented by 8 to move to the next elements in the buffer. If the counter reaches 0, meaning that the sub-FFT computation is completed, the counter is reset to the number of twiddles, and the values of ISPU_FFT_CURR_D1 and ISPU_FFT_CURR_D2 are further incremented by the number of twiddles multiplied by 8, in order to move the pointers to the elements to process for the next sub-FFT.

It is the responsibility of the user to count the number of butterfly operations ($\frac{N}{2}$) to perform in a stage and, when finished, write again registers ISPU_FFT_TWIDDLE and ISPU_FFT_CURR_D1 to prepare for the next stage.

ISPU_FFT_STATE_D (0x800D0). Here the user can find the state the accelerator is currently in, as shown in Figure 5. It is not necessary to use this register to implement the butterfly operations.

Figure 5. FFT data accelerator FSM


During the butterfly operations, in order to compute the results for the next stage, the twiddle factors $(e^{-\frac{2\pi i}{N(s)}k})$ are also needed. Similarly to the data accelerator, the twiddle table accelerator allows directly accessing the twiddle factors in the correct order, without the need to manually compute the indexes.

The twiddle table must contain the values of the cosine of an angle that ranges from 0 to $-\frac{\pi}{2}$, with step $-\frac{\pi}{N_o/2}$, for a total of $\frac{N_o}{4} + 1$ values, where N_o is the original number of points, meaning that if a real FFT is computed by packing the values in the real and imaginary parts of the complex numbers, N_o is the number of the real points.

$$N = \frac{N_o}{2}$$

An example for $N_o = 32$ is shown in Table 8.

Table 8. FFT twiddle table for 32 points

Value	Angle
1.0	0°
0.98078525	-11.25°
0.9238795	-22.5°
0.8314696	-33.75°
0.70710677	-45°
0.55557024	-56.25°
0.38268343	-67.5°
0.19509032	-78.75°
0.0	-90°

While the values have been computed for the cosine, thanks to the symmetry properties of the trigonometric functions, it is obvious how the sine can also be derived from the same table (albeit with a change of sign), thus allowing to get both the real and imaginary parts of the twiddle factors.

The twiddle factors for angles beyond $-\frac{\pi}{2}$ can also be derived from the same values available in the table, again thanks to the symmetry properties.

If a real FFT is computed by packing the values in the real and imaginary parts of the complex numbers, after the complex FFT has been computed, there is an additional stage, here called split stage, to reconstruct the real FFT results.

$$X_k = \frac{1}{2} \left[\left(X_k^{(sf)} + X_{N-k}^{(sf)*} \right) - ie^{-\frac{2\pi i}{N_o}k} \left(X_k^{(sf)} - X_{N-k}^{(sf)*} \right) \right]$$

$$X_{k + \frac{N_o}{2}} = \frac{1}{2} \left[\left(X_k^{(sf)} + X_{N-k}^{(sf)*} \right) + ie^{-\frac{2\pi i}{N_o}k} \left(X_k^{(sf)} - X_{N-k}^{(sf)*} \right) \right]$$

where $k = 0, \dots, \frac{N_o}{2} - 1$ and $X^{(sf)}$ is the result of the complex FFT.

While the data accelerator cannot be used due to the different indexing compared to the butterfly operations, the twiddle table accelerator can also be used for the split stage.

The twiddle table accelerator can be used by means of the following ISPU system registers.

ISPU_FFT_START_TW (0x80090). Here the user must write the address of the first element of the twiddle table. The address must be aligned to at least 256 bytes (bits 0 to 7 are fixed to zero). Based on the number of points, it might have to be further aligned, as shown in [Table 9](#).

Table 9. FFT twiddle table alignment

Number of points (N)	Alignment [bytes]
16	256
32	256
64	256
128	512
256	1024
512	2048
1024	4096

ISPU_FFT_END_TW (0x80094). Here the user must write the address of the last element of the twiddle table. The address must be aligned to 4 bytes (bits 0 to 1 are fixed to zero).

Bits after bit 11 are fixed to zero in this register. The high part of the address is taken from ISPU_FFT_START_TW.

ISPU_FFT_CURR_TW1 (0x80098). Here the address to the real part of the twiddle factor for the current operation is stored. At the beginning, the user must write the address of the first element of the twiddle table to this register. The address must be aligned to 4 bytes (bits 0 to 1 are fixed to zero).

Bits after bit 11 are fixed to zero in this register. The high part of the address is taken from ISPU_FFT_START_TW.

ISPU_FFT_CURR_TW2 (0x8009C). Here the address to the imaginary part of the twiddle factor for the current operation is stored. At the beginning, the user must write the address of the last element of the twiddle table to this register. The address must be aligned to 4 bytes (bits 0 to 1 are fixed to zero).

Bits after bit 11 are fixed to zero in this register. The high part of the address is taken from ISPU_FFT_START_TW.

ISPU_FFT_STRIDE_TW (0x800A0). Here, in the FFT_STRIDE_TW[8:0] field, the user must write, at the beginning of each stage, the stride to use to move to the next twiddle factor. The stride must be $\frac{N_0}{2^S}$.

If a real FFT is computed by packing the values in the real and imaginary parts of the complex numbers, the stride will have value 2 in the last stage of butterfly operations. The stride of value 1 must be used for the split stage.

To get the twiddle factors for the split stage, bit FFT_SPLIT must be set to 1, otherwise it must be set to 0.

ISPU_FFT_MATCH_TW (0x8008C). Here the user can read the currently pointed twiddle factor. For each operation, the following reads from register ISPU_FFT_MATCH_TW must be performed:

1. Read to get the real part of the twiddle factor (value pointed by ISPU_FFT_CURR_TW1).
2. Read to get the imaginary part of the twiddle factor (value pointed by ISPU_FFT_CURR_TW2 with the sign inverted if FFT_SPLIT is set to 0).

After performing the steps above, the value of ISPU_FFT_CURR_TW1 is incremented by the stride multiplied by 4 (a single-precision floating-point number is 4 bytes) and the value of ISPU_FFT_CURR_TW2 is decremented by the stride multiplied by 4 to move to the next twiddle factor. If ISPU_FFT_CURR_TW1 reaches

ISPU_FFT_END_TW, meaning that the angle has reached $-\frac{\pi}{2}$, the state machine starts looping over new states where the sign of the real part of twiddle factor is inverted:

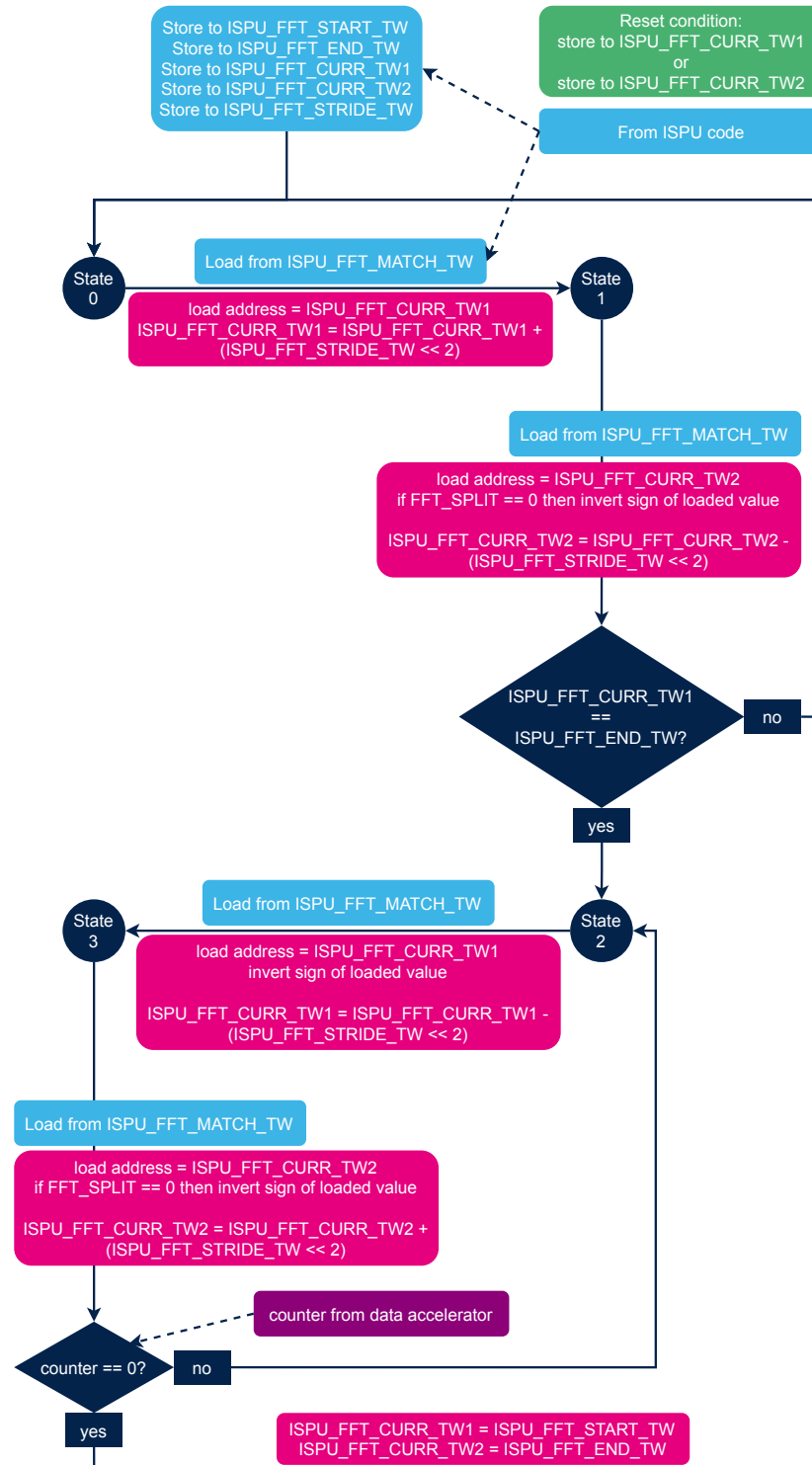
1. Read to get the real part of the twiddle factor (value pointed by ISPU_FFT_CURR_TW1 with the sign inverted).
2. Read to get the imaginary part of the twiddle factor (value pointed by ISPU_FFT_CURR_TW2 with the sign inverted if FFT_SPLIT is set to 0).

When the internal counter of the data accelerator reaches 0, the twiddle accelerator state machine is reset to the initial state, and ISPU_FFT_CURR_TW1 and ISPU_FFT_CURR_TW2 are reset, respectively, to ISPU_FFT_START_TW and ISPU_FFT_END_TW, to be ready for the next sub-FFT.

The sign of the imaginary part changes if the split bit is set. This is because the twiddle factor during this special stage is multiplied by i , which means that the imaginary part becomes real but with the opposite sign. The real part instead becomes imaginary, but with the same sign as normal.

ISPU_FFT_STATE_TW (0x800A4). Here the user can find the state the twiddle table accelerator is currently in, as shown in [Figure 6](#). It is not necessary to use this register to implement the FFT computations.

Figure 6. FFT twiddle table accelerator FSM



For a full implementation of the FFT algorithm, refer to the library and examples available in the [ISPU GitHub repository](#).

13.3 Circular buffer

The circular buffer accelerator allows implementing a circular buffer without the need to explicitly manage an index or pointer to the current element. The elements of the buffer have a size of 4 bytes.

The accelerator can be used by means of the following ISPU system registers:

- **ISPU_CIRCULAR_START** (0x800E8). Here the user must write the address of the first element of the buffer. The address must be aligned to at least 256 bytes (bits 0 to 7 are fixed to zero). If the buffer is used as a circular buffer, wrapping from the end of the buffer to the beginning, the address must also be aligned to 4096 bytes.
- **ISPU_CIRCULAR_END** (0x800EC). Here the user must write the address of the end of the buffer, that is the address immediately following the last element of the buffer. The address must be aligned to 4 bytes as every element of the buffer must be 4 bytes (bits 0 to 1 are fixed to zero). Bits after bit 11 are fixed to zero in this register. The high part of the address is taken from **ISPU_CIRCULAR_START**. This means that the buffer can be 4096 bytes long, that is, 1024 elements at the most.
- **ISPU_CIRCULAR_STRIDE** (0x800F4). Here the user must write the stride to use to access the buffer minus 1 (0 means stride of 1). The stride is expressed in number of elements, where each element is 4 bytes. The maximum stride is 8.
- **ISPU_CIRCULAR_CURR** (0x800F0). Here the user can find the address of the element of the buffer currently pointed. The address can also be written, usually to initialize it to the first element of the buffer during the initial configuration of the accelerator. The address must be aligned to 4 bytes (bits 0 to 1 are fixed to zero) as every element of the buffer must be 4 bytes. Bits after bit 11 are fixed to zero in this register. The high part of the address is taken from **ISPU_CIRCULAR_START**.
- **ISPU_CIRCULAR_MATCH** (0x800E4). Here the user can find the value of the element of the buffer currently pointed. The register can be used both for reading and writing the value. Every time an access to this register is performed (either in read or write) the accelerator moves the pointer to the next element in the buffer.

Before the buffer usage, the user must set the buffer start address in **ISPU_CIRCULAR_START**, the buffer end address in **ISPU_CIRCULAR_END**, the stride to use in **ISPU_CIRCULAR_STRIDE**, and initialize **ISPU_CIRCULAR_CURR** with the address of the buffer element to start from (usually the buffer start address, corresponding to the first element in the buffer).

In order to access the buffer the user can exploit **ISPU_CIRCULAR_MATCH** to read or write the current value of the buffer. Note that every access to **ISPU_CIRCULAR_MATCH** moves the internal pointer to the next element of the buffer. In case it is necessary to both read and write the buffer at the same time, it may be useful to perform one operation directly using **ISPU_CIRCULAR_MATCH** and the other one indirectly using **ISPU_CIRCULAR_CURR** to derive the address or index of the element in the buffer, thus avoiding incrementing two times the pointer.

The following example shows how to configure a circular buffer of length 5 and loop 3 times over it to sum the values of its elements in the variable `sum`. At the end of the 3 loops `sum` will contain the value 45 (3 times the sum of the buffer values, which is 15).

```
__attribute__((aligned(4096))) uint32_t cbuf[5] = { 1, 2, 3, 4, 5 };

*((volatile uint32_t *)ISPU_CIRCULAR_START) = (uint32_t)&cbuf[0];
*((volatile uint32_t *)ISPU_CIRCULAR_END) = (uint32_t)&cbuf[sizeof(cbuf) / sizeof(cbuf[0])];
*((volatile uint32_t *)ISPU_CIRCULAR_STRIDE) = 0; // stride of 1
*((volatile uint32_t *)ISPU_CIRCULAR_CURR) = (uint32_t)&cbuf[0];

uint32_t sum = 0;
for (uint8_t i = 0; i < 3; i++) { // loop 3 times over the entire buffer
    for (uint8_t j = 0; j < sizeof(cbuf) / sizeof(cbuf[0]); j++) // loop over the buffer
        sum += *((volatile uint32_t *)ISPU_CIRCULAR_MATCH); // add the current buffer value
}
```

As shown in the example, the buffer start address can be obtained by getting the address of the first element of the buffer. The end address can be obtained by getting the address of the element of the buffer indexed with the length of the buffer itself, which is the first non-valid element of the buffer.

13.4 Square root

The square root accelerator allows computing the square root of a single-precision floating-point number.

The accelerator can be used by writing the input value in the ISPU system register ISPU_SQRT_I (0x800F8). The computation takes 3 clock cycles, after which the result is available in the ISPU system register ISPU_SQRT_O (0x800FC).

```
*((volatile float *)ISPU_SQRT_I) = input;
asm("nop");
asm("nop");
asm("nop");
float res = *((volatile float *)ISPU_SQRT_O)
```

As shown in the example above, it is recommended to insert 3 `nop` instructions after writing the input and before reading the output, to ensure to wait for the result to be computed. Alternatively, instead of the `nop` instructions, other useful operations can be inserted. However, note that if C instructions are used, the compiler could rearrange the instructions, thus creating issues.

13.5 Accumulate and multiply

The accumulate and multiply (ACMS) instruction allows performing an addition on an accumulator and a multiplication in one single clock cycle. The instruction has the following syntax:

```
acms    %ry, %rx
```

where `ry` and `rx` are two registers containing the operands. The two operands must be two single-precision floating-point numbers.

The instruction first adds `r1` to `r0`. Register `r0` acts as the accumulator. Register `r1` contains the result of the multiplication from the previous execution of the `acms` instruction or another value set by the user if it is the first execution of the instruction. Then it multiplies the two operands `ry` and `rx` and puts the result in `r1`.

Before starting a series of accumulate and multiply there are two options:

- Set both `r0` and `r1` to 0 so that the first addition will not have any effect.

```
movw %r0, 0
movw %r1, 0
```

In this case, the `acms` will have to be used starting from the first operands.

- Perform the first two multiplications and set the results in `r0` and `r1` so that the first addition will already accumulate the first two results.

```
fmuls %r0, %r2 ; assuming that r0 and r2 contain the first operands
fmuls %r1, %r3 ; assuming that r1 and r3 contain the second operands
```

In this case, the `acms` will have to be used starting from the third operands.

In any case, after the last usage of the `acms` instruction, an addition will have to be performed to add to `r0` the result of the last multiplication available in `r1`.

```
fadds %r0, %r1
```

The following example shows how to compute the squared norm of an array of 4 elements. At the end of the code `r0` will contain the result, with a value of 25.0 (2.5 squared times 4).

```
; define an array with all elements equal to 2.5
movw %r0, 0x40200000 ; v[0] = 2.5
movw %r1, 0x40200000 ; v[1] = 2.5
movw %r2, 0x40200000 ; v[2] = 2.5
movw %r3, 0x40200000 ; v[3] = 2.5

fmuls %r0, %r0      ; compute square of v[0] in r0 (accumulator)
fmuls %r1, %r1      ; compute square of v[1] in r1
acms %r2, %r2        ; add square of v[1] to accumulator and compute square of v[2]
acms %r3, %r3        ; add square of v[2] to accumulator and compute square of v[3]
fadds %r0, %r1       ; add square of v[3] to accumulator
```

14 Memory access

The ISPU is supported by two separate RAMs, one of 32 KB, usually occupied by code, and one of 56 KB (8 KB + 8 KB + 8 KB + 32 KB), normally used to store data.

In addition to being accessible from the ISPU, the contents of the two RAMs can also be accessed from the device interface (SPI or I3C) using specific procedures. As explained in [Section 15: Program loading](#), this is necessary in order to load the binary of the program to the RAM.

Note: *Before using the RAMs, the first step is to power them on. This can be performed by writing 1 to both `FIFO_EN` and `ISPU_EN` in the `CTRL3 (0x12)` register, after which a minimum wait time of 2 ms is required before proceeding. Both bits must be set for the ISPU.*

In order to write to the RAM, these steps must be followed:

1. Write 1 to `ISPU_RAM_ACCESS_IF` in the `RAM_ACCESS (0x01)` register to enable access to the RAM from the device interface.
2. Wait at least 0.45 ms.
3. Configure the `ROM_SEL` bit in the `ISPU_MEM_SEL (0x07)` register to select one of the two RAMs.
 - a. Writing 0 to `ROM_SEL` selects the 56 KB data RAM.
 - b. Writing 1 to `ROM_SEL` selects the 32 KB code RAM.
4. Write 0 to `READ_RAM` in the `ISPU_MEM_SEL` register to enable writing.
5. Write the address that the data must be written to in the `ISPU_MEM_ADDR` registers.
 - a. The `ISPU_MEM_ADDR_2 (0x08)` register must contain the most significant byte of the address.
 - b. The `ISPU_MEM_ADDR_1 (0x09)` register must contain the least significant byte of the address.
6. Write the data at the selected location by writing it in the `ISPU_MEM_DATA_1 (0x0A)` register. If multiple writes are performed without setting a new address, each new byte is written to the next location (the address is automatically incremented).
7. Write 0 to `ISPU_RAM_ACCESS_IF` in the `RAM_ACCESS` register to disable access to the RAM from the device interface.

Note: *While the `ROM_SEL` bit is used to select the RAM to access, it can also be seen as part of the address. In this perspective, register `ISPU_MEM_SEL` can be seen as the highest part of the address (except for the addition of the `READ_RAM` bit). Referring to [Section 3: Memory mapping](#), it can be observed that the 32 KB code RAM starts from address 0x40000, which corresponds exactly to setting the `ROM_SEL` bit to 1 and the `ISPU_MEM_ADDR` registers to 0.*

Note that `ISPU_MEM_SEL`, `ISPU_MEM_ADDR`, and `ISPU_MEM_DATA` are part of the ISPU register page, so, in order to perform the above procedures, access to the ISPU register page must be enabled as explained in [Section 4: ISPU registers](#). This can be done at the same time as enabling RAM access in step 1.

Here is an example of the write procedure implemented in C code:

```
write(0x01, 0xC0); // enable RAM access and ISPU register page access

delay(0.45); // delay in ms

union mem {
    uint8_t bytes[4];
    uint32_t value;
} address;

address.value = addr; // extract address bytes

write(0x07, address.bytes[2]); // set write mode and address most significant byte
write(0x08, address.bytes[1]); // set address middle byte
write(0x09, address.bytes[0]); // set address least significant byte

write_mul(0x0A, data_buf, data_len); // write data

write(0x01, 0x00); // disable RAM access and ISPU register page access
```

where `addr` is a `uint32_t` variable containing the address value, `write` is a function implementing the write of a single byte to the device over the SPI or I3C interface, `write_mul` is a similar function, but supporting the write of multiple bytes in one single transaction, `data_buf` is the buffer containing the data to write, and `data_len` is the length in bytes of the data to write.

In order to read from the RAM, these steps must be followed:

1. Write 1 to ISPU_RAM_ACCESS_IF in the RAM_ACCESS (0x01) register to enable access to the RAM from the device interface.
2. Wait at least 0.45 ms.
3. Configure the ROM_SEL bit in the ISPU_MEM_SEL (0x07) register to select one of the two RAMs.
 - a. Writing 0 to ROM_SEL selects the 56 KB data RAM.
 - b. Writing 1 to ROM_SEL selects the 32 KB code RAM.
4. Write 1 to READ_RAM in the ISPU_MEM_SEL register to enable reading.
5. Write the address that the data must be read from in the ISPU_MEM_ADDR registers.
 - a. The ISPU_MEM_ADDR_2 (0x08) register must contain the most significant byte of the address.
 - b. The ISPU_MEM_ADDR_1 (0x09) register must contain the least significant byte of the address.
6. Read the data at the selected location by reading it from the ISPU_MEM_DATA registers. Every time the device will read 2 bytes, which will be available, in order, in ISPU_MEM_DATA_1 (0x0A) and ISPU_MEM_DATA_2 (0x0B). If multiple reads are performed without setting a new address, each new byte is read from the next location (the address is automatically incremented).
7. Write 0 to ISPU_RAM_ACCESS_IF in the RAM_ACCESS register to disable access to the RAM from the device interface.

Note: *Every time a new address is set in the ISPU_MEM_ADDR registers, the first two bytes must be discarded with a read from ISPU_MEM_DATA_1.*

Note: *If the read starts from an odd address, the actual read, after discarding the two bytes, should be performed using register ISPU_MEM_DATA_2 instead of ISPU_MEM_DATA_1. This is because when an odd address is set, the bytes read from ISPU_MEM_DATA_1 will start from the previous even address.*

Here is an example of the read procedure implemented in C code:

```
write(0x1, 0xC0); // enable RAM access and ISPU register page access

delay(0.45); // delay in ms

union mem {
    uint8_t bytes[4];
    uint32_t value;
} address;

address.value = addr; // extract address bytes

write(0x07, address.bytes[2] | 0x40); // set read mode and address most significant byte
write(0x08, address.bytes[1]); // set address middle byte
write(0x09, address.bytes[0]); // set address least significant byte

uint8_t discard[2];
read(0x0A, discard, 2); // discard first 2 bytes

if (address.value % 2 == 0)
    read(0x0A, data_buf, data_len); // read data from even address
else
    read(0x0B, data_buf, data_len); // read data from odd address
```

where `addr` is a `uint32_t` variable containing the address value, `read` is a function implementing the read of multiple bytes in one single transaction from the device over the SPI or I3C interface, `data_buf` is the buffer where the read data will be stored, and `data_len` is the number of bytes to read.

Note: *The main usage of the above procedures is to load the ISPU program and, optionally, read it back to check if it was correctly loaded. It is possible to access the RAM at runtime (after the ISPU boot), but with the following limitation: if the sensor is going from idle to continuous mode or vice versa, or going from sleep to sampling or vice versa when configured in burst mode, the RAM cannot be accessed. If a RAM access is attempted during these transitions, the correct functioning of the device is not guaranteed.*

Note: *If the ISPU clock is enabled (sensor not set in idle mode in register CTRL1 and CLK_DIS set to 0 in ISPU_CONFIG), but not active (ISPU sleeping), accesses to the RAM from the interface have the effect of activating the clock. The ISPU does not wake up (no code is executed), but, if the ISPU sleep signal is routed to the INT2 pin (INT2_SLEEP_ISPU bit in register INT_CTRL1 set to 1), then the INT2 pin briefly changes state during the RAM access. If the external microcontroller is configured to be interrupted by signals coming from pin INT2 of the sensor, it is advisable to temporarily disable either the reception of the interrupt at the microcontroller level or the routing of the sleep signal to the INT2 pin at the sensor level. This is recommended as continuous interrupts to the microcontroller could disrupt the ongoing communication with the sensor interface.*

15 Program loading

Once a program has been compiled using the ISPU toolchain, it must be loaded in the RAM of the device before performing the boot procedure described in [Section 16: Boot procedure](#).

Before loading the program in the device, it is recommended to perform a software reset of the ISPU by setting the SW_RESET_ISPU bit of the ISPU_CTRL2 (0x19) register to 1 and then setting it back to 0 after waiting at least 100 µs. This step is necessary if the ISPU was previously booted. It resets the ISPU core and registers.

The primary output of the toolchain is a binary ELF (executable and linkable format) file containing the binary of the program. However, it is possible to convert the ELF file to the Motorola S-record (SREC) format, which is text-based and much easier to parse. This conversion can be done using the toolchain's `objcopy` tool specifying the option `-O srec`.

The SREC file contains different types of lines, but the lines of interest are those starting with "S2". These lines contain the actual data to write to the device and have the following format:

	type	count	address	data	checksum
S	2	NN	AAAAAA	D[0] D[0] D[1] D[1] ... D[N-1] D[N-1]	CC

- **S** is the first character for every line in an SREC file.
- The **type** field indicates the record type, in this case (type = 2) a data record with a 24-bit address.
- The **count** field is the number of bytes in the record (including **address**, **data**, and **checksum**), expressed as one byte in hexadecimal format.
- The **address** field is expressed as a 24-bit value in hexadecimal format and represents the address the **data** must be written to. RAM addresses for the ISPU utilize only 19 of the 24 bits, the remaining bits are set to zero. The most significant bit (of the 19 bits) represents the RAM selection and can be mapped to the ROM_SEL bit in the ISPU_MEM_SEL (0x07) register. Bits 18 and 17 are always 0. The remaining 16 bits represent the address that can be set in the ISPU_MEM_ADDR_2 (0x08) (most significant byte) and ISPU_MEM_ADDR_1 (0x09) (least significant byte) registers.
- The **data** field is also expressed in hexadecimal format and can vary in size depending on the **count** field. These bytes must be loaded sequentially to the device RAM starting from the **address**.
- The **checksum** field contains a one byte checksum of the **count**, **address**, and **data** fields, expressed in hexadecimal format.

In order to load the program it is then possible to parse the SREC file and, for each "S2" line, select the RAM to write to, set the address specified, and write the data bytes included in the line, all while following the write procedure described in [Section 14: Memory access](#). Note that the address automatic increment can be leveraged to reduce the number of writes to the device.

Note: *The toolchain provided by STMicroelectronics allows converting the SREC format into ready-to-use formats that are supported by STMicroelectronics tools and that can be directly integrated in custom firmware. These formats contain the sequence of operations to load the program from the device interface (SPI or I3C).*

16 Boot procedure

After the program has been loaded in the device RAM, the following procedure must be followed in order to boot the ISPU core:

1. Set the ISPU rate to 0 Hz by writing ISPU_RATE_[3:0] in the ISPU_CTRL1 (0x18) register to 0x0.
2. Enable the ISPU clock by writing the CLK_DIS bit of the ISPU_CONFIG (0x02) register to 0.
3. Disable the ISPU reset by writing the RST_N bit of the ISPU_CONFIG register to 1.
4. Power on the accelerometer sensor by writing ODR_XL_[3:0] in the CTRL1 (0x10) register to a value between 0x2 and 0x7 (a valid rate different from idle mode).
5. Wait at least 7 ms for the accelerometer sensor to power on.
6. Wait for the end of the boot before any further sensor configuration.

Note that steps 2 and 3 can be performed with one single write to the ISPU_CONFIG register. ISPU_CONFIG is part of the ISPU register page.

The boot time depends on the loaded program, so, in order to know if the boot has ended, the external microcontroller can poll the BOOT_END bit in the ISPU_STATUS (0x04 / 0x80804) register and wait for its value to become 1 (boot completed). The ISPU_STATUS register is part of the ISPU register page. The boot is to be considered completed when the boot code has been executed and the program is ready to process data samples. Since the boot code is written by the user and can change, it is up to the user to set the BOOT_END bit to 1 from the ISPU code once the boot code has been successfully executed.

As an alternative to polling the BOOT_END bit, the external microcontroller can implement a delay to wait for a predefined time. However, in order to define the value of this delay, the boot time should be estimated in advance during the development of the ISPU program. For example, using the templates and examples provided by STMicroelectronics, a wait of 2 ms is sufficient. The BOOT_END bit must be in any case set to 1 from the ISPU, otherwise the ISPU does not wake up from sleep after stopping the clock as explained in [Section 2: Processing cycle](#).

Note: *If a reboot of the ISPU is needed, the software reset described in [Section 15: Program loading](#) must be performed before the boot procedure, even if no new program needs to be loaded.*

The boot procedure can be automatically generated by the toolchain provided by STMicroelectronics.

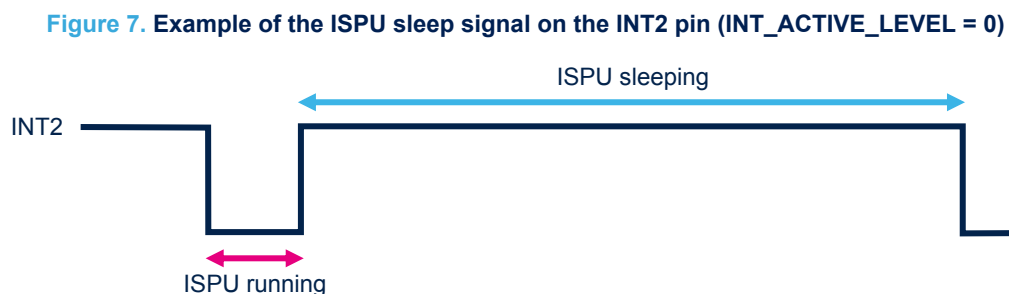
17 Performance

The ISPU has a fixed clock of 40.96 MHz, thus reaching 40.96 MIPS and 40.96 MFLOPS. It scores 1.95/MHz in the CoreMark benchmark.

The key to maximizing the performance of an ISPU program is to measure the execution time of the program in order to identify bottlenecks and track the effect of any change applied to the code, and optimize the code considering the specificities of the ISPU architecture and using the available hardware accelerators.

17.1 Measuring execution time

The INT2 pin can be used for monitoring the time the ISPU is running. The ISPU sleep signal can be routed to the INT2 pin of the device by setting the INT2_SLEEP_ISPU bit in the INT_CTRL1 (0x0C) register to 1. When this functionality is enabled, the INT2 line is set to the inactive level while the ISPU is running and to the active level while the ISPU is sleeping. Using the signal produced on the INT2 pin, it is then possible to measure the total execution time of the ISPU processing. This can be achieved by measuring the time that the signal remains continuously at the inactive level, for example using a logic analyzer or a timer on the external microcontroller. [Figure 7](#) shows an example of the signal produced on the INT2 pin after enabling the INT2_SLEEP_ISPU bit.



If the INT2 pin cannot be used, the sleep status of the ISPU can also be monitored by reading the bit ISPU_CORE_SLEEP in register DEVICE_STATUS (0x21). When the bit has a value of 1, the ISPU is sleeping.

As explained in [Section 2: Processing cycle](#), evaluating the execution time is important to make sure that the processing can keep up with the generation of the data to process. Additionally, it makes it possible to evaluate the impact of optimizations to improve the performance of the processing logic, which, even if it is already fast enough to process data at the configured data rate, can still be improved to reduce the amount of time the ISPU is running and thus the power consumption.

Another way to evaluate the performance of the implemented code is to configure the interrupt pins in pulsed mode and control either or both interrupt pins directly. This allows measuring the performance of specific portions of code instead of the whole processing logic. This can be achieved, for example, by setting an interrupt pin to the inactive level just before the portion of code of interest and setting it back to the active level just after. The execution time for the portion of code of interest can then be obtained by measuring the time that the interrupt signal remains continuously at the inactive level (using a logic analyzer or a timer on the external microcontroller). For details on how to configure and control the interrupt pins, refer to [Section 12: Outbound interrupts](#).

17.2 Code optimization

Here is a list of points to consider for optimizing code for the ISPU:

- Review the instruction set available in [Table 10 of Appendix A: ISPU instruction set](#). Any operation not implemented in the hardware should be avoided whenever possible. For example, the following operations are compiled to use a software implementation (incurring slower performance and higher code size):
 - Integer division. One example where it can be avoided is when wrapping an index of a buffer. Instead of using the modulus operator %, check if the index has exceeded the buffer length and set it to its initial value if it has.
 - 32-bit integer multiplication. If sufficient for the application, use 16-bit variables.
 - Floating-point division. In case of division by constant values, use multiplication for the inverse value instead.
 - Floating-point comparisons. If needed, the optimized functions provided in the template on the [ISPU GitHub repository](#) can be used.
 - Any double-precision floating-point operation. The `double` data type has no hardware support and should be avoided unless strictly necessary.
- Make use, whenever possible, of the hardware accelerators described in [Section 13: Hardware accelerators](#).
- The [ISPU GitHub repository](#) provides common functions already optimized for the ISPU architecture and exploiting the hardware accelerators. If possible, reuse what is already available instead of implementing everything from scratch.

Appendix A ISPU instruction set

Table 10. ISPU instruction set

Group	Acronym	Description
Move	LD	Load data from memory
	ST	Store data in memory
	MOV	Move data between registers
Arithmetic	UEXT	Unsigned-value extension
	SEXT	Signed-value extension
	ADD	Integer addition
	SUB	Integer subtraction
	MUL	Integer multiplication
	MAX	Unsigned integer maximum
	MIN	Unsigned integer minimum
Floating	FADD	Floating-point single-precision addition
	FSUB	Floating-point single-precision subtraction
	FMUL	Floating-point single-precision multiplication
	FMAX	Floating-point single-precision maximum
	FMIN	Floating-point single-precision minimum
	ACMS	Floating-point single-precision accumulate and multiply
Shift	SRL	Shift-right logical
	SRA	Shift-right arithmetic
	SLA	Shift-left arithmetic
	ROTC	Rotate with carry
	ROT	Rotate left or right
Logical	AND	Bitwise AND
	OR	Bitwise OR
	XOR	Bitwise XOR
	CMP	Compare
Bit manipulation	BSET	Bit set
	BCLEAR	Bit clear
	BEXT	Bit extract
	BINV	Bit invert
Branch	JPD	Jump displacement
	JPI	Jump from immediate
	JPR	Jump from register
	JLR	Jump from register and link
	JLI	Jump from immediate and link
	RET	Return
	RFE	Return from exception
Special	NOP	No operation

Revision history

Table 11. Document revision history

Date	Version	Changes
05-Aug-2026	1	Initial release

Contents

1	ISPU.....	2
2	Processing cycle	4
3	Memory mapping	6
4	ISPU registers	7
5	External registers.....	8
6	Sensor data	9
7	Configuration.....	10
8	Time-related inputs	11
9	Outputs	12
10	FIFO.....	13
10.1	FIFO write	13
10.2	FIFO read.....	13
11	Inbound interrupts.....	15
11.1	Boot.....	16
11.2	Traps.....	16
11.3	Data valid / high-priority interrupt	16
11.4	Overrun	16
11.5	Low-priority interrupts	17
11.6	Algorithms	17
12	Outbound interrupts	19
13	Hardware accelerators	22
13.1	Zero-overhead loop.....	22
13.2	FFT (Fast Fourier Transform)	23
13.3	Circular buffer	31
13.4	Square root	32
13.5	Accumulate and multiply	32
14	Memory access.....	33
15	Program loading.....	36
16	Boot procedure.....	37
17	Performance.....	38
17.1	Measuring execution time	38
17.2	Code optimization	39
Appendix A	ISPU instruction set.....	40

Revision history	41
List of tables	44
List of figures.....	45

List of tables

Table 1.	ISPU rate selection	4
Table 2.	Accelerometer mode selection	5
Table 3.	Sleep counter resolution	11
Table 4.	ISPU interrupt requests.	15
Table 5.	ISPU interrupt modes	19
Table 6.	FFT_N_BRA[2:0] values	23
Table 7.	FFT data buffer alignment	26
Table 8.	FFT twiddle table for 32 points.	28
Table 9.	FFT twiddle table alignment.	28
Table 10.	ISPU instruction set	40
Table 11.	Document revision history	41

List of figures

Figure 1.	Sensor with ISPU core.	2
Figure 2.	ISPU recommended processing cycle	20
Figure 3.	FFT bit reverse addressing accelerator FSM	24
Figure 4.	FFT butterfly operation (DIT)	25
Figure 5.	FFT data accelerator FSM	27
Figure 6.	FFT twiddle table accelerator FSM	30
Figure 7.	Example of the ISPU sleep signal on the INT2 pin (INT_ACTIVE_LEVEL = 0)	38

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice.

In the event of any conflict between the provisions of this document and the provisions of any contractual arrangement in force between the purchasers and ST, the provisions of such contractual arrangement shall prevail.

The purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgment.

The purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of the purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

If the purchasers identify an ST product that meets their functional and performance requirements but that is not designated for the purchasers' market segment, the purchasers shall contact ST for more information.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2026 STMicroelectronics – All rights reserved