MGS2401 User Manual

A microcontroller by

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Mindgrove Silicon's MGS2401 is a low-power, high-performance microcontroller designed for IoT and embedded applications. It operates at a clock frequency of 700 MHz, with an I/O voltage of 1.8 V and a core voltage of 0.9 V.

The device includes a security complex that accelerates cryptographic algorithms such as AES (128, 192, and 256), 2048-bit RSA, and SHA-256. It also provides secure storage for sensitive data, such as cryptographic keys, in a write-protected and read-restricted One-Time Programmable (OTP) memory.

MGS2401 features a 128 KB of on-chip SRAM and 8 KB of Boot ROM. External memory of up to 512 MB can be added to MGS2401 if required.

1.1 Who Should Read This Manual?

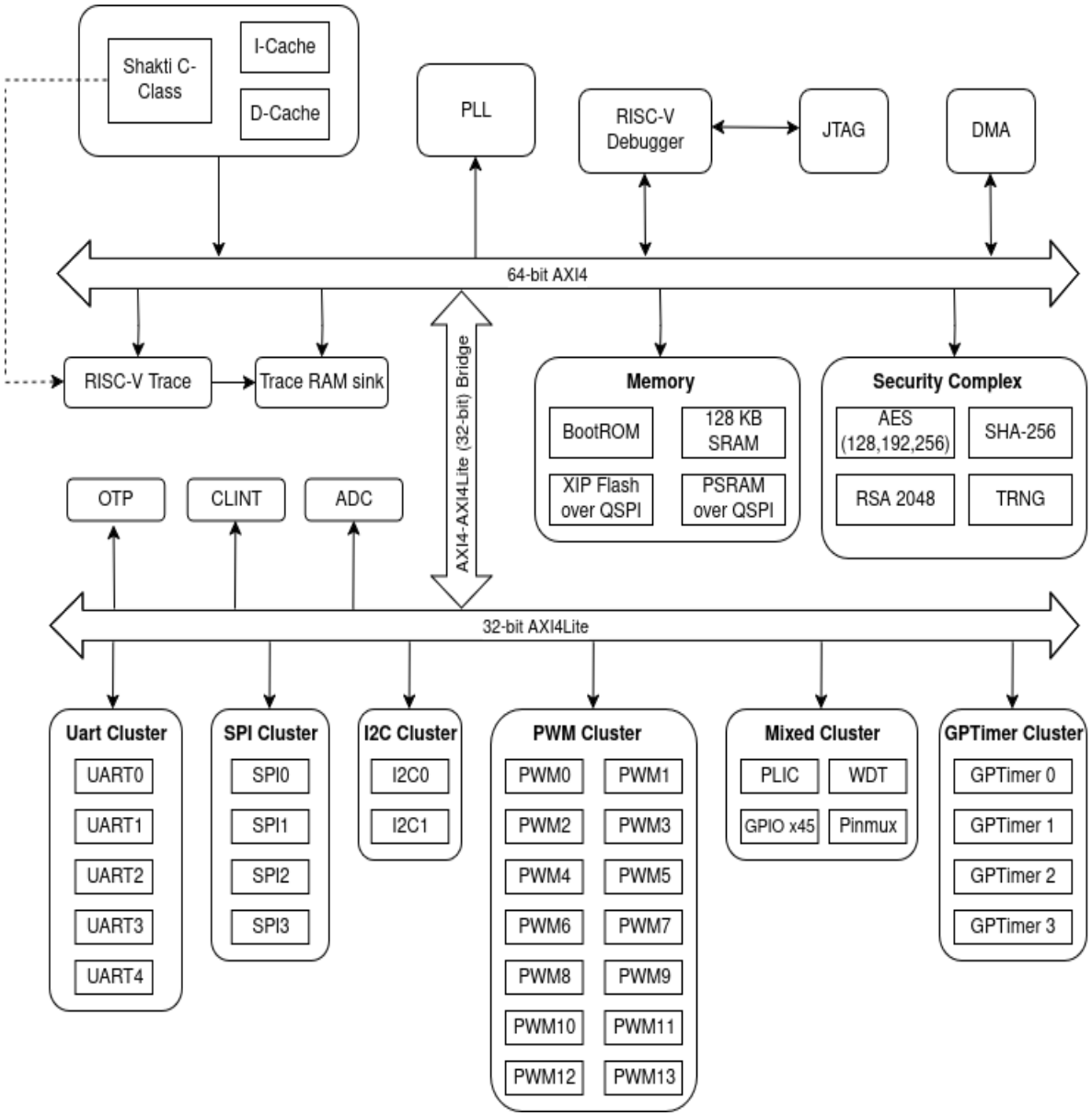
- This manual is intended for developers and engineers who are using Mindgrove Silicon's MGS2401 *MGS2401-B144C-B*.
- It describes all aspects of the MGS2401 that developers and engineers need to know for application development.
- It provides detailed descriptions of all peripherals in the MGS2401, including their configuration, control and status registers, and API references.
- It also includes example applications using the APIs to help developers understand the behavior of peripherals and accelerators.

Note

This manual is suitable for developers, engineers, students, and hobbyists interested in working with microcontrollers.

1.2 Overview of MGS2401

The figure below shows the overall block diagram of the Mindgrove Silicon MGS2401. A summary of the components is provided in the *Features and Descriptions* table.



1.3 Shakti C-Class Core

The Shakti C-Class core is a configurable, commercial-grade processor supporting the standard RV64IMAFDC instruction set architecture (ISA) extensions. It features a six-stage in-order pipeline targeting mid-range embedded applications and edge computing systems.

The core includes 16 KB of I-Cache and 16 KB of D-Cache, both 4-way set associative, with separate fully associative instruction and data TLBs. A simple two-state branch

predictor enhances performance.

It supports multiple privilege levels, including User Mode, Supervisor Mode, and Machine Mode. The core can run real-time operating systems such as Zephyr, FreeRTOS, and NuttX. Memory Management Units (MMUs) enable virtual memory functionality.

Fabricated using advanced technology nodes (22nm FinFET and 180nm CMOS), the core was developed by RISE Lab at IIT Madras and released as open-source. It achieves a performance rating of 1.78 DMIPS/MHz. Hardware features include IEEE-754 compliant single and double precision FPUs with low-overhead sequential operations, and a six-entry return address stack to improve subroutine call efficiency.

The flexibility of the Shakti C-Class core allows it to be customized for various applications, including IoT devices, industrial controllers, motor control systems and storage controllers.

1.4 Features and Descriptions

Table 1.4.1: Features and Descriptions

Feature	Description
RISC-V CORE	Shakti C-Class processor with RV64IMAFDC ISA extensions, running at 700 MHz with a core voltage of 0.9 V.
General Purpose Input/Output (GPIO)	45 GPIO pins, each capable of operating up to 1 MHz, with programmable interrupt support.
Pulse Width Modulation (PWM)	14 PWM pins supporting configurable period, duty cycle, and deadband delay.
PINMUX	30 pin-muxed pins including GPIO, PWM, UART, SPI, and JTAG.
Serial Peripheral Interface (SPI)	4 SPI ports supporting data transfers up to 35 Mbps in Master/Slave mode (Mode 0 and Mode 3), each with programmable interrupts.

continues on next page

Table 1.4.1 – continued from previous page

Feature	Description
Quad Serial Peripheral Interface (QSPI)	2 QSPI ports for high-speed external memory, supporting up to 70 MHz with programmable interrupts.
Universal Asynchronous Receiver / Transmitter (UART)	5 UART ports with programmable interrupt support.
Inter-Integrated Circuit Interface (I2C)	2 I2C ports supporting standard mode (up to 400 kHz) and fast mode (up to 1 MHz).
Analog to Digital Conversion (ADC)	12-bit SAR ADC with 8 channels, supporting up to 5 MSPS.
JTAG Debugger	Hardware debug module accessible via JTAG, compliant with RISC-V Debug Specification v0.13.
Instruction Trace (I-Trace)	Hardware implementation of RISC-V Efficient Trace Spec v2.0. Encodes instruction flow in dedicated I-Trace RAM for later decoding.
Watch-Dog Timer (WDTimer)	Allows software-triggered system resets without physical intervention.
General-Purpose Timer (GPTimer)	4 GPTimers supporting multiple modes for accurate delays and interval measurement.
Platform Level Interrupt Controller (PLIC)	Aggregates interrupts from all peripherals and notifies the system when an interrupt occurs.
Rivest Shamir Adleman (RSA)	Single instance of 2048-bit hardware-accelerated RSA for digital signatures, secure key exchange, and encryption/decryption.

continues on next page

Table 1.4.1 – continued from previous page

Feature	Description
Advanced Encryption Standard (AES)	Hardware-accelerated AES supporting 128, 192, and 256-bit keys, with CBC, CFB, OFB, and CTR modes.
Secure Hashing Algorithm (SHA)	Hardware-accelerated SHA-256 for efficient message verification, ensuring data authenticity and application integrity.
One-Time Programmable Memory (OTP Memory)	32 Kbit OTP memory for secure storage of cryptographic keys and secure boot.
True Random Number Generator (TRNG)	NIST SP800-90C compliant TRNG generating unpredictable numbers for cryptographic key generation.

Chapter 2. Boot Configuration

Mindgrove Silicon's MGS2401 provides Secure-Boot capability. The boot configuration is stored in the on-chip Boot-ROM.

2.1 Boot Modes Supported

1. Secure-Boot with QSPI0/QSPI1.
2. Normal-Boot with QSPI0/QSPI1.
3. Parking Loop if QSPI0/QSPI1 is not available.

2.2 Pre-Requisites for Boot

For successful booting of the software application, in modes 1 and 2, the following are expected to be present.

1. QSPI Flash connected to **QSPI0** or **QSPI1** interface, programmed with a valid software application. The flash must support **XIP (Execute In Place)** mode. This is required because QSPI0 or QSPI1 switches to XIP mode where program execution begins directly from flash memory instead of copying the code to RAM.
2. **UART0** is set as the default serial output and cannot be changed by the user. The serial terminal baudrate must be set to **115200** to view the boot output.

2.3 Boot Process

Upon powering on or resetting, MGS2401 initiates the boot procedure, which focuses on security validation and flash setup.

1. **JTAG Lock Status**
    - The device reads the OTP memory location at address `0x3028` to determine the status of the `JTAG_LOCK` bit.
    - If the bit is **set**, hardware locks the JTAG interface to prevent external debug access.

- If the bit is **not set**, JTAG remains unlocked for development or programming purposes.

2. **QSPI Flash Detection**
   - The system checks for Serial Flash Discoverable Parameters (**SFDP**) first on **QSPI0**.
   - If detected, **QSPI0** is used as the boot source.
   - If QSPI0 does not present a valid SFDP response, **QSPI1** is probed.
   - If QSPI1 is valid, it becomes the boot source.

   If neither QSPI interface presents a valid flash device:
   - The system sets `BOOT_STATUS: NO_FLASH`
   - A PWM status LED blinks at **500 ms** intervals
   - The system halts.
3. **QSPI Configuration**
   - After flash detection, the bootloader checks for a custom QSPI configuration header at flash address `0x08`.
   - If a valid configuration is present, custom settings are applied to the QSPI controller registers (**DCR**, **CR**, **CCR**).
   - If absent, default **XIP (execute-in-place)** settings are loaded for compatibility.
4. **Application Size Check**
   - The bootloader reads the application image size from flash address `0x00`.
   - If the application size is greater than **16 MB**, the system sets `BOOT_STATUS: INVALID APP LEN`, blinks the status LED at **500 ms** intervals, and halts the system.
5. **Application Code Validation**
   - The application code at its expected entry address is checked for validity.
   - If the entry point contains an invalid pattern, the bootloader signals `BOOT_STATUS: NO_APPLN`, blinks the status LED at **500 ms** intervals, and halts.
6. **Secure Boot Enablement**
   - If application headers are valid, the bootloader checks secure boot enable bits in OTP.
   - If secure boot is enabled, the secure boot process starts.
   - Otherwise, the application is executed directly from flash.

7. **Key Discovery and Public Key Validation**
   - The bootloader reads public key data from OTP memory (`OTP:0x3008`) searching for a valid cryptographic key.

   If no valid public key is found:
   - The system signals `ERR_CODE: NO PUB KEY`
   - The status LED blinks at **250 ms** intervals
   - The core stalls and debug access is blocked.

   If the key is invalid:
   - The system signals `ERR_CODE: INVALID PUB KEY`
   - The core stalls and debug access is blocked.
8. **Signature and Hash Verification**
   - The bootloader checks **PKCS #1 v1.5** padding in the decrypted signature.
   - Padding failure triggers `ERR_CODE: RSA_PKCS_DECRYPT`, LED blinking, and a core halt.
   - The runtime **SHA hash** is compared with the signed hash.
   - A mismatch produces `ERR_CODE: SHA_MISMATCH`, LED blinking, and a core halt.
9. **Secure or Non-Secure Boot Success**

   On successful verification:
   - The bootloader sets `GPIO6=LOW` and `GPIO11=GPIO15=HIGH` to indicate **Secure Boot Success**.
   - The application execution begins.

   If secure boot is **not enabled**:
   - The application executes immediately after simple validation.
   - GPIO indication: `GPIO6=LOW, GPIO11=HIGH`.
10. **Exception and Trap Handling**
    - If an exception occurs during application execution, the trap handler logs the `mcause` and `mepc` registers.
    - The PWM status LED blinks pulses corresponding to the **mcause code**.
    - The system then enters a while loop and halts.

2.4 Flowchart

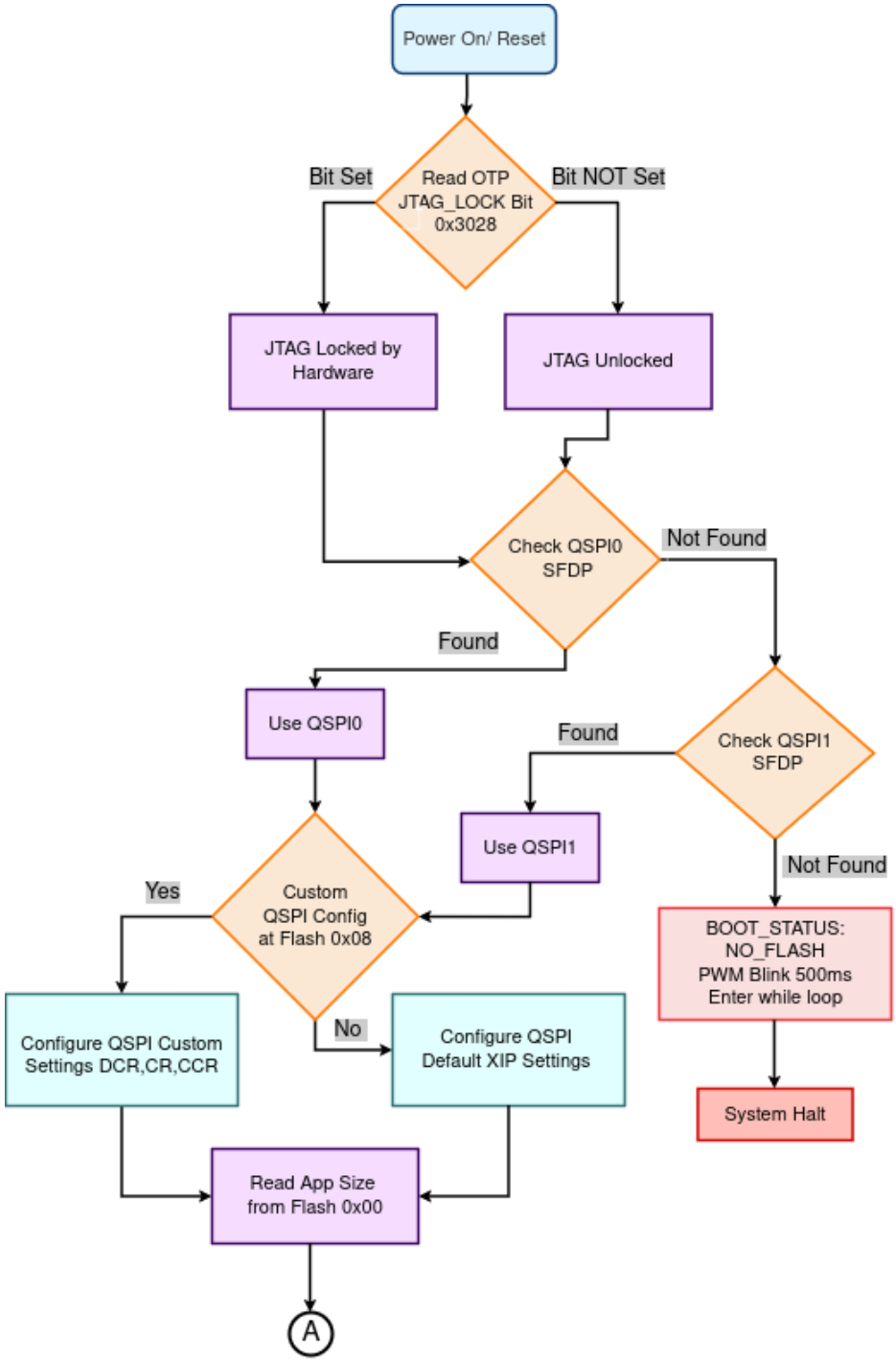


Fig. 2.4.1: Secure Boot Flow 1

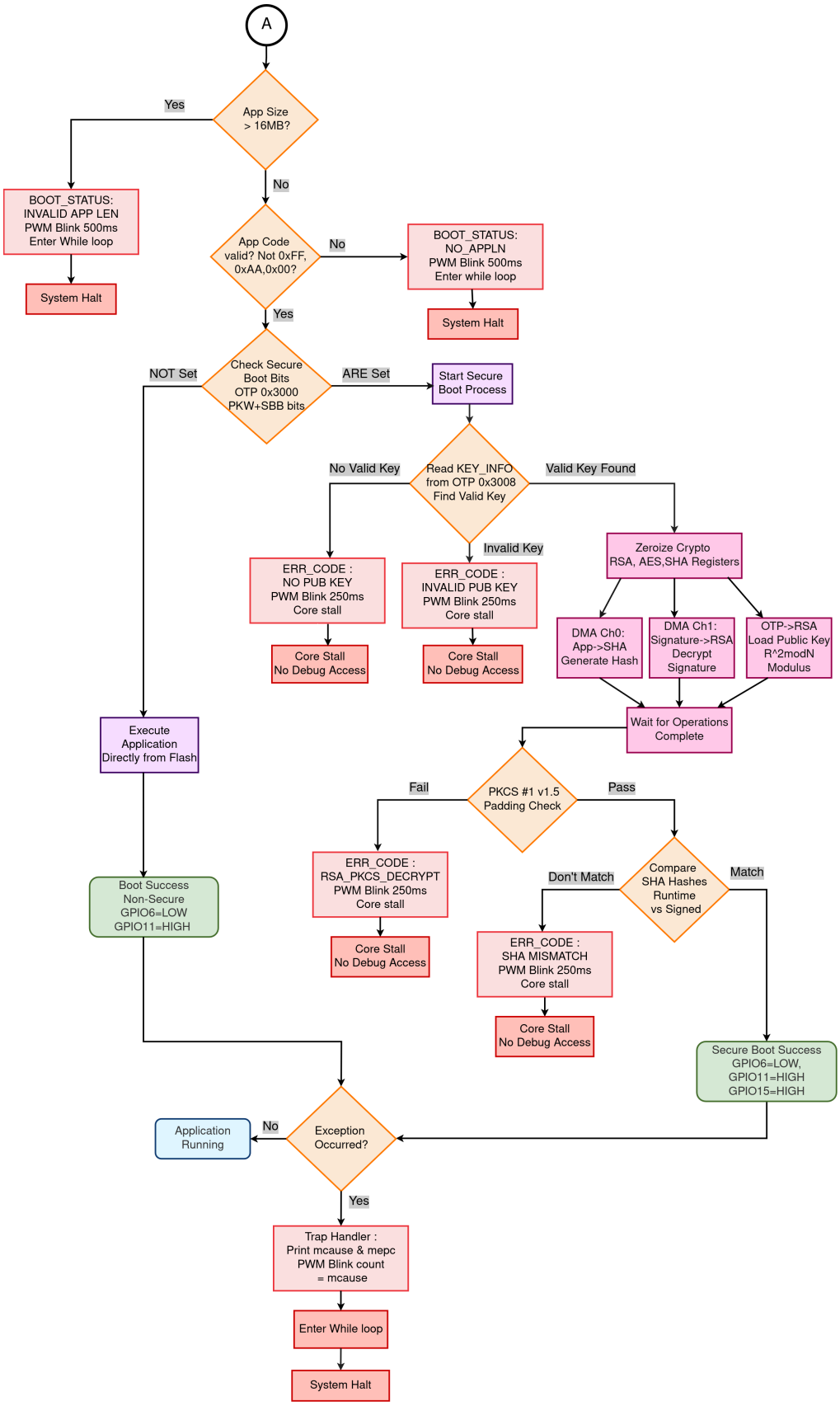


Fig. 2.4.2: Secure Boot Flow 2

Chapter 3. Setting up

3.1 Evaluation Board Setup

3.1.1 Board Layout Overview

The evaluation board layout is shown below for reference.

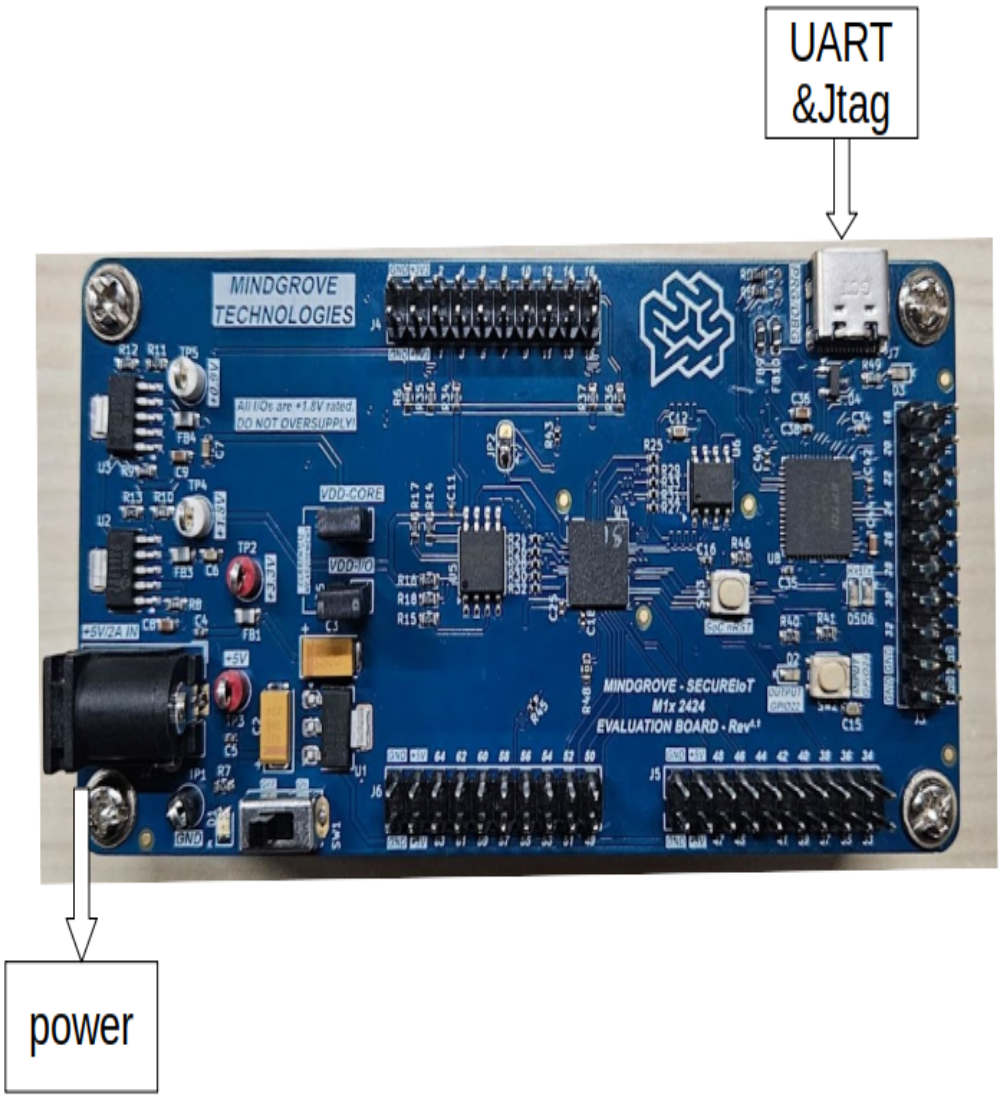


Fig. 3.1.1: Evaluation board layout

3.1.2 Power Supply

Power can be provided by using the adapter shown below.

Fig. 3.1.2: Power adapter

3.1.3 JTAG/USB Connection

Use a **Type-C to Type-A cable** to connect the evaluation board to the host system. This connection provides both **JTAG** and **serial port** access.

Fig. 3.1.3: Type-C to Type-A cable connection

3.1.4 Wiring

Connect the appropriate wires as shown in the below figure.

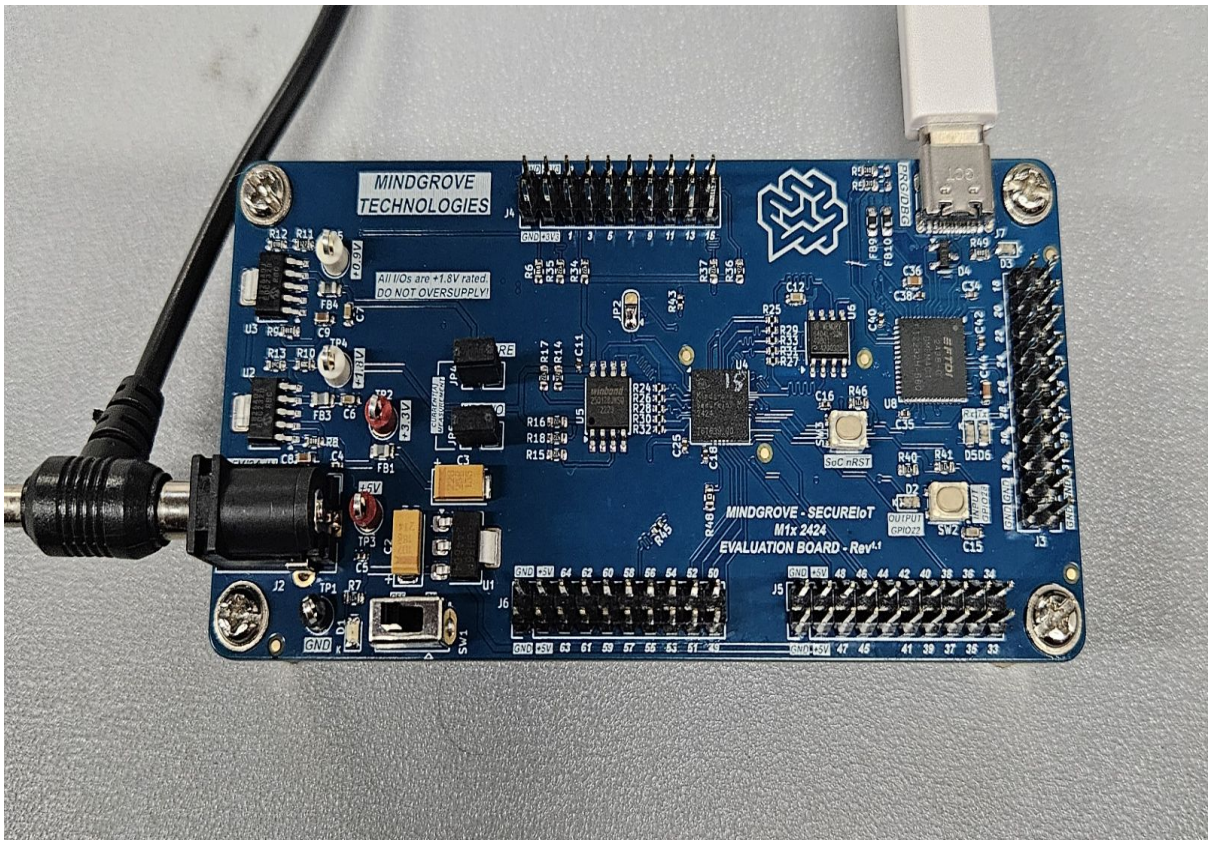


Fig. 3.1.4: Wiring connections

3.1.5 Power On

After connecting all cables:

1. Turn on the board using the power switch.
2. Confirm the **power LED** is glowing (board is ON).

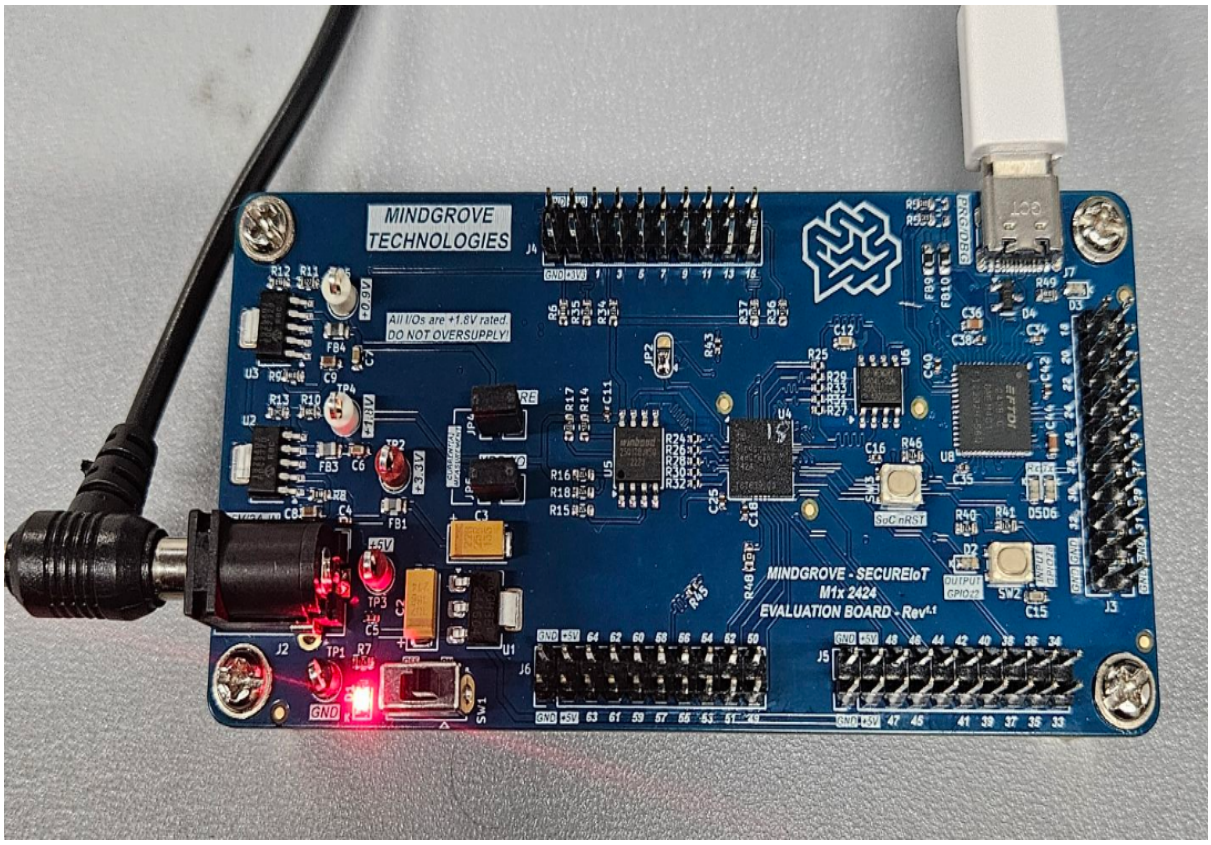


Fig. 3.1.5: Power on indication

3.1.6 Verify JTAG/Serial Connection

When the Type-A end of the Type-C to Type-A cable is connected to your system, a second LED will glow, confirming **JTAG and serial port are active**.

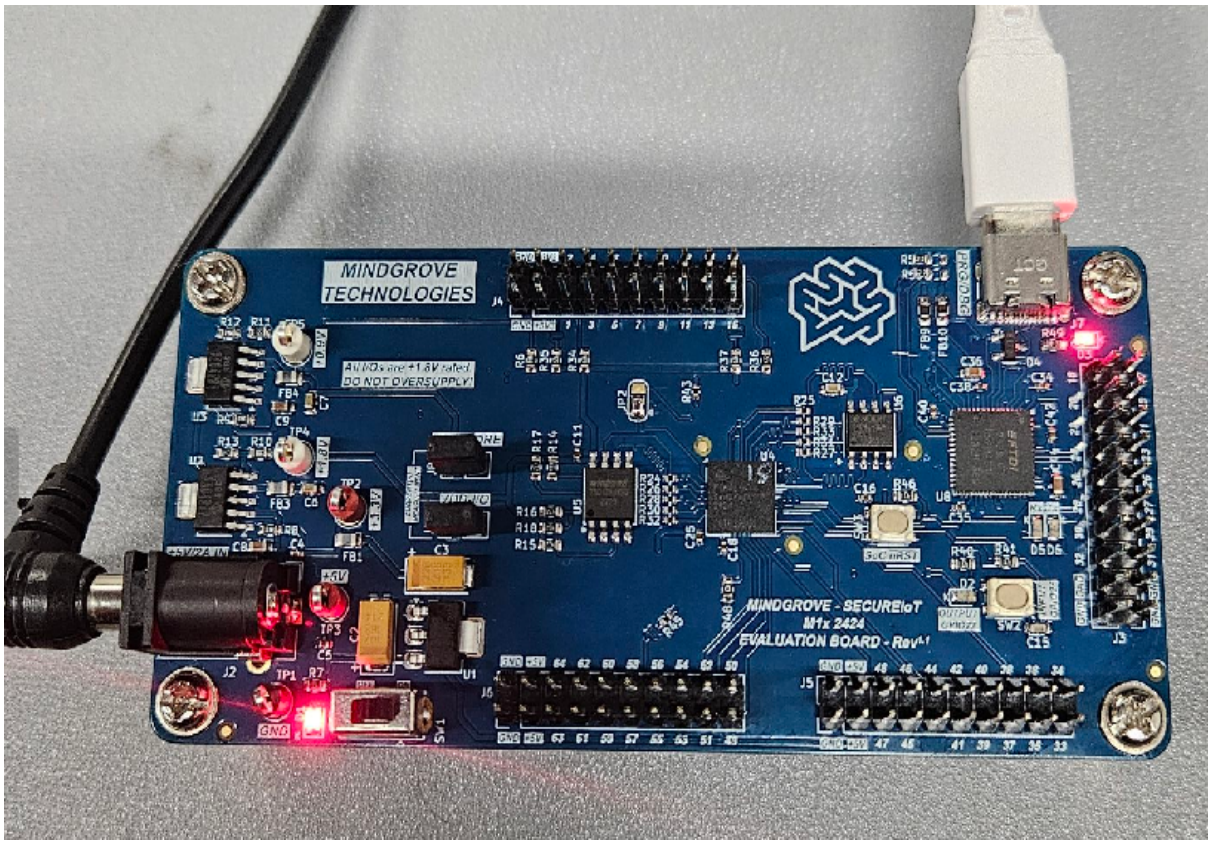


Fig. 3.1.6: JTAG and serial connection active

3.2 Software Setup and Troubleshooting

After finishing all physical connections, you will need to install software on your host computer to communicate with the device.

3.2.1 Prerequisites

You will need a serial terminal program to view boot messages and a debugger for programming and debugging. We recommend **GtkTerm** for the serial terminal.

You can install **GtkTerm** using the following command on Ubuntu 22.04:

```
sudo apt install gtkterm libusb-0.1 libftdi-dev libftdi1 libftdi1-dev␣
↪libftdi1-2 -y
```

3.2.2 Configuring GtkTerm

Once installed, you must configure GtkTerm to connect to the device with the correct settings.

1. Open GtkTerm from your terminal by running the command:

```
gtkterm
```

2. From the GtkTerm menu, select `Configuration -> Port`.
3. In the **Port** dropdown, select the correct serial device. This is typically `/dev/ttyUSB0` or `/dev/ttyUSB1` if you are using a USB-to-serial adapter.
4. Set the **Baud Rate** to `115200`.
5. Ensure other settings are at their default values (Parity: None, Bits: 8, Stopbits: 1).
6. Click **OK** to save the configuration.

Note

You can also use other serial terminal programs like `minicom` (`sudo apt install minicom`).

3.2.3 Output

As soon as you switch on the board or press reset, you will be able to see the following boot message.

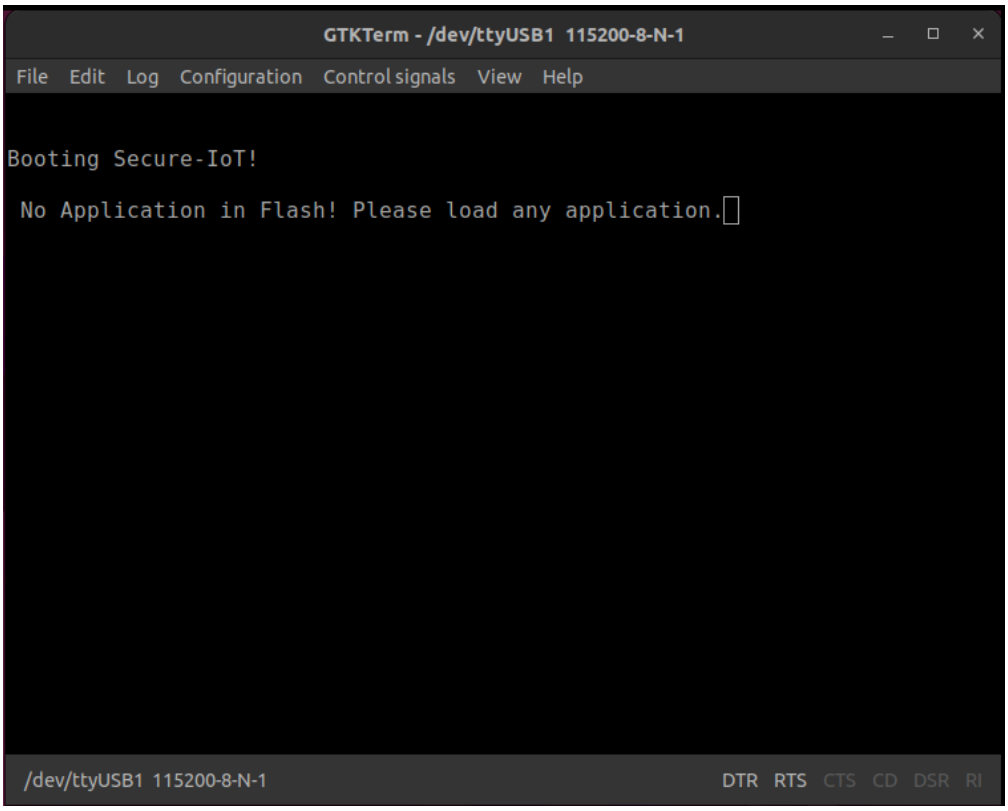


Fig. 3.2.1: Serial output

3.2.4 Troubleshooting Common Errors

Error: "Cannot open /dev/ttyS0: Permission denied"

This is a common error that occurs when your user account does not have permission to access the serial port hardware.

To fix this, you must add your user to the `dialout` group, which owns the serial port devices.

1. Use the following command to add your current user to the *dialout* group.

```
sudo usermod -aG dialout $USER
```

2. **Log out and log back in.** This step is to ensure the group changes take effect.

After logging back in, you should be able to start *gtkterm* without the permission error

The following segments will provide you the information about building applications for SecureIoT.

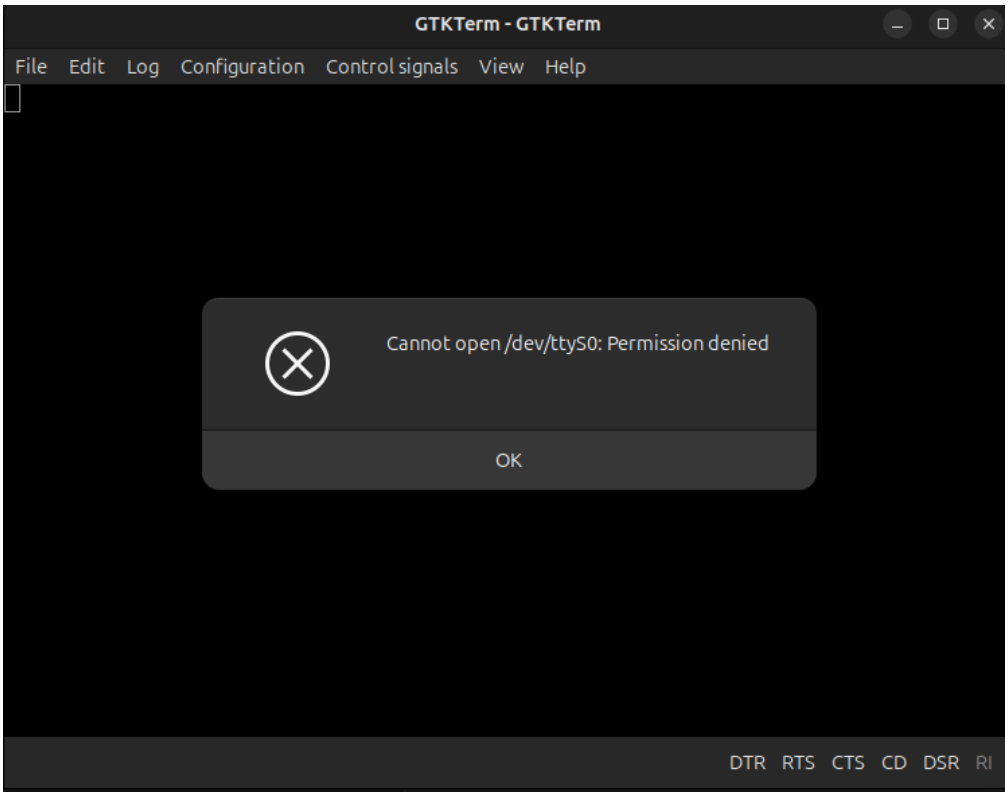


Fig. 3.2.2: GtkTerm permission denied error

3.3 Toolchain Setup

This document describes the steps to setup a RISC-V GNU toolchain for building Secure-IoT applications. It is recommended to download the pre-built binaries rather than build from source.

3.3.1 Using pre-built binaries

Download the pre-built binaries for Ubuntu 22.04 or Ubuntu 24.04 and extract them. Copy the extracted `riscv` folder to `/opt/riscv`:

```
tar -xvzf gcc15.2_24.04.tar.gz
cd opt
sudo cp -r riscv /opt/
```

Replace `gcc15.2_24.04.tar.gz` with the name of the downloaded tarball in the above commands. Then follow the instructions starting from the “Add to PATH” section to

finish the installation and test the toolchain.

3.3.2 Building from source

Warning

The source code and build files will occupy more than 10 GB of space on disk. The build process will take a long time and a powerful machine with at least 32 GB of RAM and 8 CPU cores is recommended.

Install dependencies

```
sudo apt-get install autoconf automake autotools-dev curl python3␣
↪python3-pip python3-tomli libmpc-dev libmpfr-dev libgmp-dev gawk␣
↪build-essential bison flex texinfo gperf libtool patchutils bc␣
↪zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev␣
↪libslirp-dev libncurses-dev
```

Note

The above command is for Ubuntu 22.04 or 24.04. For later Ubuntu versions and other Linux distributions, please refer to the official documentation of the distribution for equivalent packages.

Get the source code

It is recommended to download the source tarball provided by Mindgrove here. If so, you may skip the remainder of this section.

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
git checkout 19dd147bdcfaa2bead5bca408a5e788a6032099f
```

Checkout the supported versions of GCC and Newlib:

```
cd gcc
git checkout releases/gcc-15.2.0
```

(continues on next page)

(continued from previous page)

```
cd ../newlib
git checkout newlib-4.5.0
cd ../gdb
git checkout gdb-15.2-release
cd ..
```

Note

Unlike typical Linux environments, Secure-IoT does not use GLibC as the default C library. Instead, it uses Newlib for some standard C library functions while the rest are provided by the Secure-IoT SDK.

Build and install

From the root directory of the toolchain, run the following commands:

```
./configure --prefix=/opt/riscv --with-arch=rv64imafdc --with-
↪abi=lp64d --enable-multilib --with-cmodel=medany
sudo make -j 8
sudo make install
```

3.3.3 Add to PATH

Add the following line to your shell configuration file (e.g., ~/.bashrc or ~/.zshrc).

```
export PATH=/opt/riscv/bin:$PATH
```

3.3.4 Test

Open a new terminal and verify that the toolchain is installed correctly by running the following command:

```
riscv64-unknown-elf-gcc -v
```

You should see output similar to the following with the correct version number:

```
Using built-in specs.
COLLECT_GCC=riscv64-unknown-elf-gcc
COLLECT_LTO_WRAPPER=/opt/riscv/libexec/gcc/riscv64-unknown-elf/15.2.0/
 ↪lto-wrapper
Target: riscv64-unknown-elf
Configured with: /tools/mg-rv-gnu-toolchain/gcc/configure --
 ↪target=riscv64-unknown-elf --prefix=/opt/riscv --disable-shared --
 ↪disable-threads --enable-languages=c,c++ --with-pkgversion= --with-
 ↪system-zlib --enable-tls --with-newlib --with-sysroot=/opt/riscv/
 ↪riscv64-unknown-elf --with-native-system-header-dir=/include --
 ↪disable-libmudflap --disable-libssp --disable-libquadmath --disable-
 ↪libgomp --disable-nls --disable-tm-clone-registry --src=.././gcc --
 ↪enable-multilib --with-abi=lp64d --with-arch=rv64imafdc --with-isa-
 ↪spec=20191213 'CFLAGS_FOR_TARGET=-Os    -mcmodel=medany' 'CXXFLAGS_
 ↪FOR_TARGET=-Os    -mcmodel=medany'
Thread model: single
Supported LTO compression algorithms: zlib
gcc version 15.2.0 ()
...
```

At this point, GDB will have also been installed, which you can run with the following command:

```
riscv64-unknown-elf-gdb
```

Chapter 4. Connecting the Debugger

4.1 Installation Steps

Make sure to install the following packages before setting up the debugger.

```
sudo apt install gtkterm libusb-1.0-0-dev libftdi1-dev libftdi1␣
↪libftdi-dev -y
```

Download the OpenOCD from the link provided below:

```
https://github.com/Mindgrove-Technologies/riscv-openocd/releases/
↪download/Mindgrove-OpenOCD-V1.3/mind-openocd-v1_3
```

Provide the necessary permission to run the executable:

```
cd ~/Downloads/
chmod +x mind-openocd-v1_3
sudo cp mind-openocd-v1_3 /usr/local/bin/.
```

The `FT2232H-JTAG.cfg` configuration file is located at the following path:

`SecureIoT_Apps/scripts/FT2232H-JTAG.cfg`

After successfully installing the OpenOCD executable, type the following command to ensure that it is working properly:

```
sudo mind-openocd-v1_3 -f SecureIoT_Apps/scripts/FT2232H-JTAG.cfg
```

Expected Output:

```
Open On-Chip Debugger 0.12.0+dev-00495-ge31f10b79-dirty (2025-08-11-09:44) MINDGROVE RELEASE V1.3
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : clock speed 10000 kHz
Info : JTAG tap: riscv.cpu tap/device found: 0x100039d3 (mfg: 0x4e9 (IIT Madras), part: 0x0003, ver: 0x1)
Warn : [riscv.cpu.0] The target's debug bus (DMI) address width is lower than the minimum:
Warn : [riscv.cpu.0]  found dtmcs.abits = 6; minimum is abits = 7.
Info : [riscv.cpu.0] datacount=12 progbufsize=0
Warn : [riscv.cpu.0] We won't be able to execute fence instructions on this target. Memory may not always appear consistent. (progbufsize=0, impebreak=0)
Info : [riscv.cpu.0] Vector support with vlenb=0
Info : [riscv.cpu.0] S?aia detected with IMSIC
Info : [riscv.cpu.0] Examined RISC-V core
Info : [riscv.cpu.0]  XLEN=64, misa=0x800000000014112d
[riscv.cpu.0] Target successfully examined.
Info : [riscv.cpu.0] Examination succeed
Info : [riscv.cpu.0] starting gdb server on 3333
Info : Listening on port 3333 for gdb connections
riscv.cpu.0 halted due to undefined.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

Fig. 4.1.1: OpenOCD successfully connected to target

4.2 Troubleshooting

If OpenOCD reports errors such as:

```
Info : Listening on port 3333 for gdb connections
Error: Target not examined yet
Error: [riscv.cpu.0] Unsupported DTM version: -1
Error: [riscv.cpu.0] Could not identify target type.
```

These messages typically indicate that the JTAG interface cannot communicate with the target board. This can happen for one of the following reasons:

1. **UART/JTAG cable not connected**

   Ensure that the JTAG or UART cable is firmly connected to both the host computer and the target board.

2. **Board not powered on**

   Verify that the board's power cable is properly connected and that the board is switched **ON**.

3. **Incorrect configuration or OpenOCD version**

   Double-check that you are using the correct configuration file (`.cfg`) and the compatible OpenOCD build for your board.

4. **USB driver reset required**

   Please **reboot the system** and **reconnect the JTAG/USB cable** after the reboot. This step helps in reinitializing the USB interface, as some Linux systems may have unstable or incomplete `lsusb` driver handling for FTDI-based JTAG interfaces.

Chapter 5. Building projects

5.1 Executing from Terminal

5.1.1 Git cloning

First git-clone the SecureIoT_Apps repository with the help of the following command.

```
git clone git@github.com:Mindgrove-Technologies/SecureIoT_Apps.git
```

5.1.2 Creating Application

- Open the cloned repository.
- You can see the **src/** directory where you can find the sample applications.
- If you need to write a “blink” application, first create a *blink/* directory inside the **src/** directory. Within this directory, you should then create a *main.c* file in **SecureIoT_Apps/src/blink/** where you can write your code.
- Connect 3 LEDs to gpio 0,1,2.

5.1.3 Compiling

To compile the application for OCM, navigate back to the **SecureIoT_Apps/** directory and run the following command:

```
make blink
```

To compile the application for FLASH, navigate back to the **SecureIoT_Apps/** directory and run the following command:

```
make blink FLASH_LINK=y
```

Your **.elf** will be generated inside **work_dir/_tmp_blink_output_/**

5.1.4 Run the application directly using openocd without using debugger

To run an application in OCM using openocd, navigate back to the **SecureIoT_Apps/** directory and run the following command:

```
make blink RUN
```

This will upload your application into RAM and will start executing.

5.1.5 Run the application in FLASH using openocd

To run an application in FLASH using openocd, navigate back to the **SecureIoT_Apps/** directory and run the following command:

```
make blink FLASH
```

This will upload your application into FLASH. To execute your application press button 28 on the EVALUATION board.

5.1.6 Running the executable from Terminal in OCM

- Open 2 terminals,

Terminal 1

- Open **SecureIoT_Apps/scripts/** and host OpenOCD using the following command:

```
sudo ./mind-openocd-v1_3 -f FT2232H-JTAG.cfg
```

Termminal 2

- Open **SecureIoT_Apps/** and connect **GDB** using the following command:

```
riscv64-unknown-elf-gdb
```

- Set the timeout to unlimited and connect the gdb to the OpenOCD once the gdb is opened.

```
set remotetimeout unlimited
target remote :3333
```

- Upload the file into the fpga with **file** command followed by the directory of the executable

```
file work_dir/_tmp_blink_output_/blink.elf
load
```

- Run the executable in SecureIoT with the help of the following command

```
c
```

To enable the debugging

To enable the debug symbols for debugging, enable DEBUG flag while compiling.

```
make blink DEBUG=y
```

5.2 Executing from IDE

5.2.1 Creating a new Project

Open Platform IO from VS Code by clicking on the PlatformIO. (In Visual Studio Code, the PlatformIO icon is located on the left sidebar of the interface, within the activity bar, appearing as a dedicated icon once you have installed the PlatformIO extension).

- Click on New Project for creating a New Project.
- Fill the appropriate project name in the Project Wizard.
- Select the Board as Mindgrove Silicon's SecureIoT SoC, by Shakti C-Class(Mindgrove Technologies Private Limited)
- Choose the framework to be SecureIoT_Apps and Proceed by selecting "Finish"

Alternately, to open an existing project, Click on "Open Project" in the PlatformIO home screen and Choose the desired project.

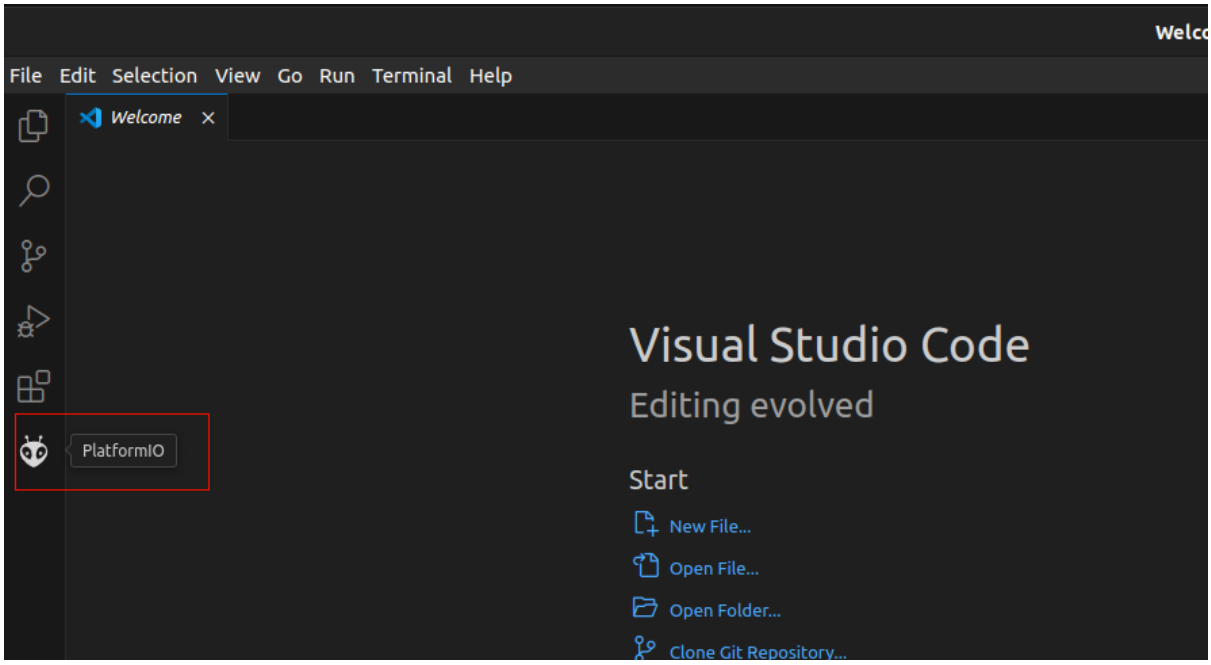


Fig. 5.2.1: Launching PlatformIO from VS Code

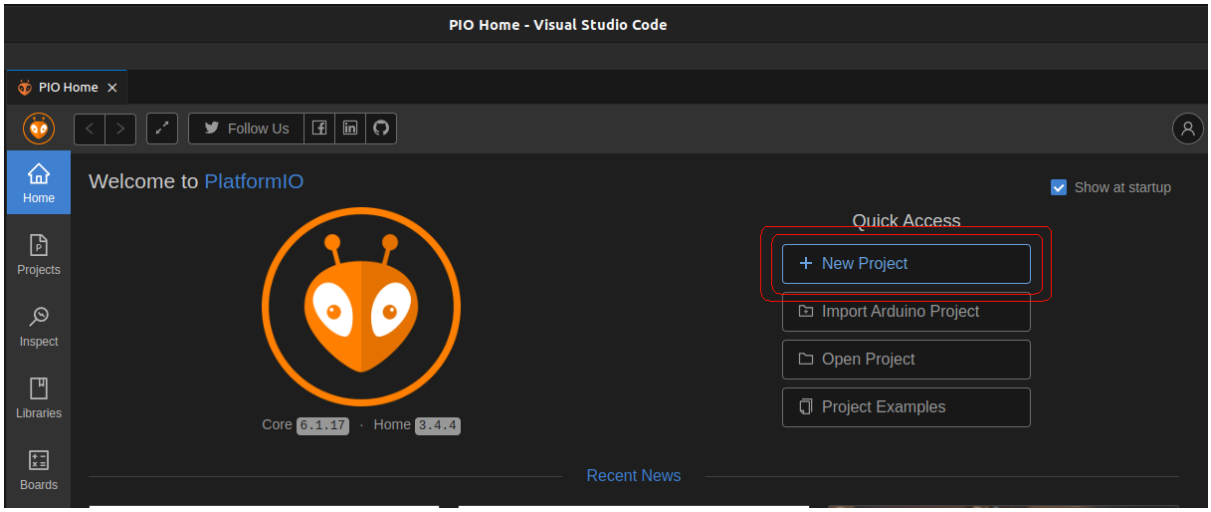


Fig. 5.2.2: Creating a new project in PlatformIO

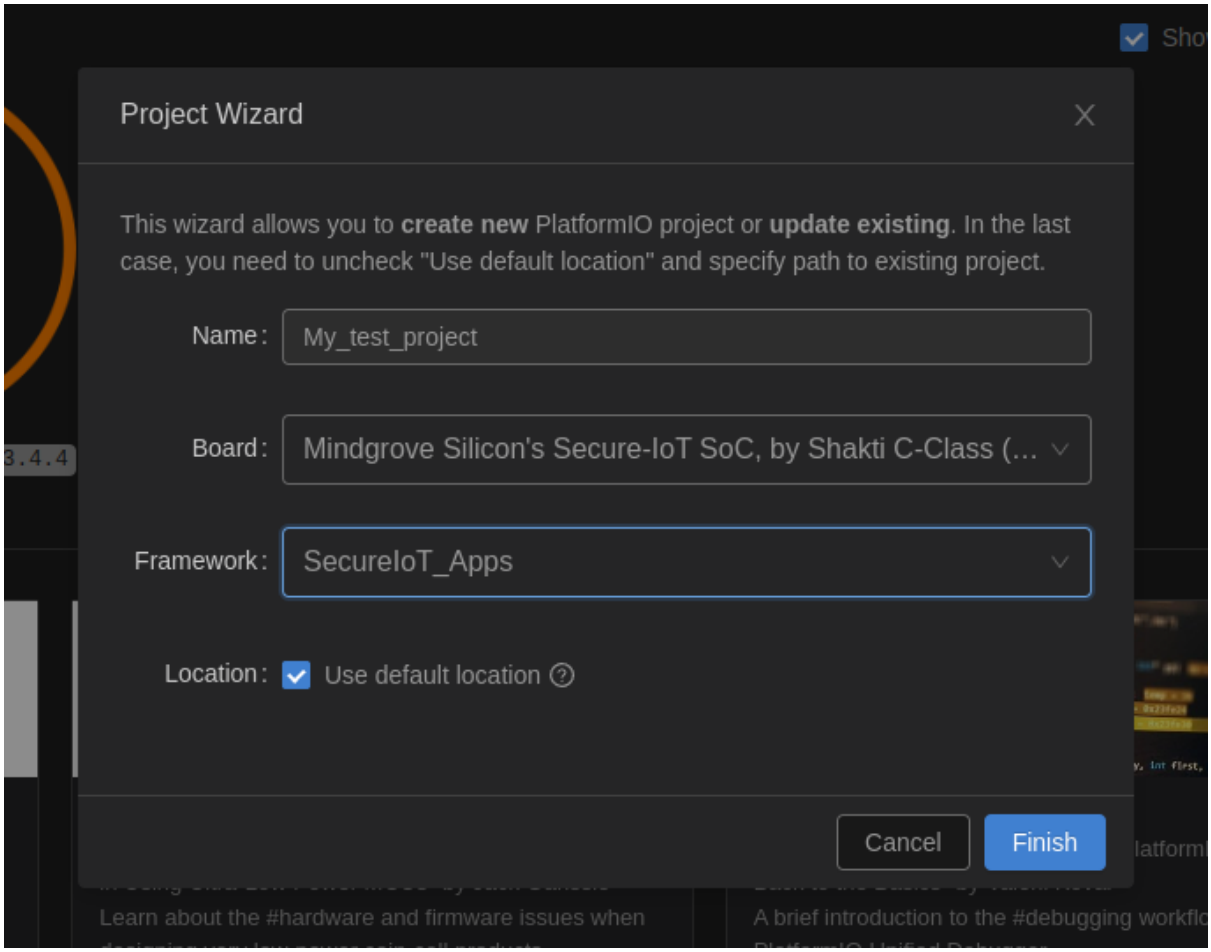


Fig. 5.2.3: Project configuration details

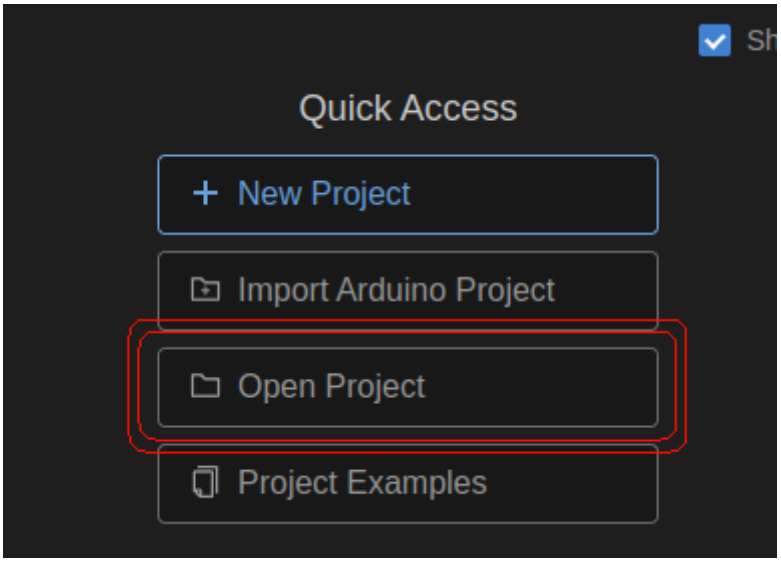


Fig. 5.2.4: Opening an existing project

5.2.2 Setting up an application

Once PlatformIO Opens up, create the desired application under the src directory. Here we have taken a simple AES encryption example

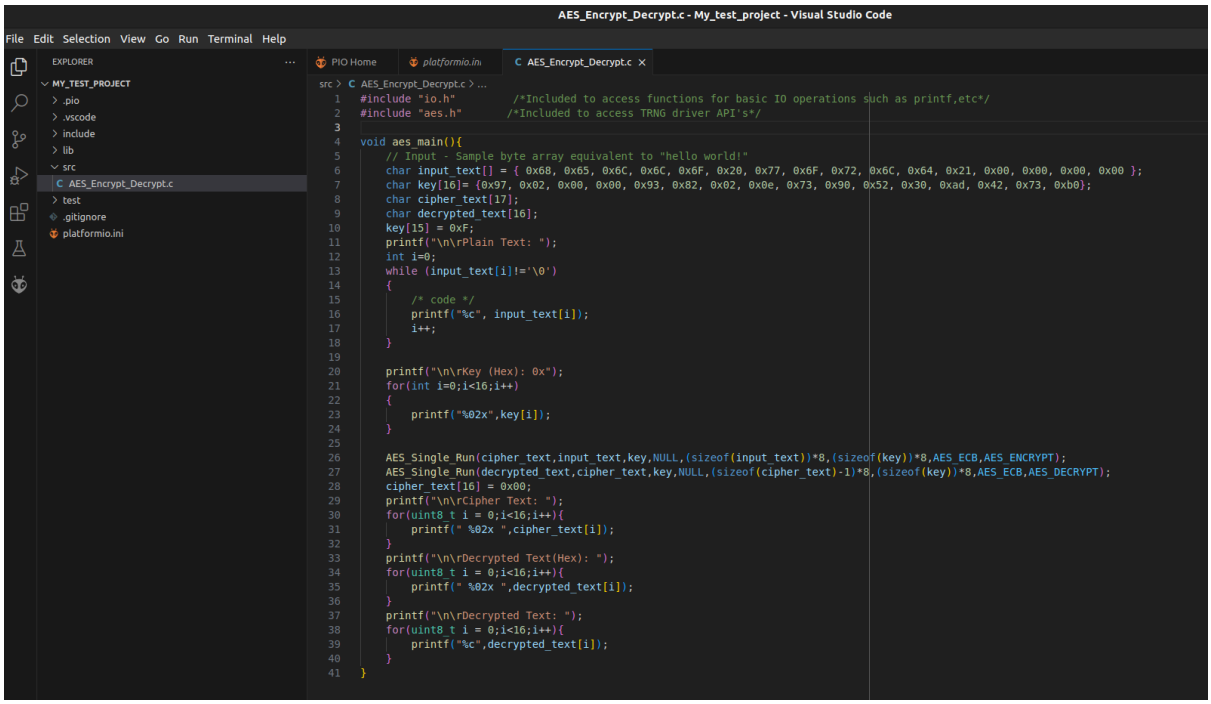


Fig. 5.2.5: Creating an application under src directory

5.2.3 Configuring PlatformIO.ini

A .ini file is a plain text file that stores configuration settings for software applications. The configuration settings for our application will be present in the parent folder with the name “platformio.ini”. It is crucial to make sure that the platformio.ini file is configured properly in order to execute the application.

Open the file and update the contents based on the sample shared below.

```
[env:secure-iot]
platform = mindgrove
framework = SecureIoT_Apps
board = secure-iot
```

(continues on next page)

(continued from previous page)

```
build_type=debug
monitor_speed = 115200
monitor_port=/dev/ttyUSB1
upload_protocol = ft2232
debug_tool=ft2232
debug_speed=14000
debug_build_flags=-O0 -ggdb3 -g3
```

- `build_type`

  Corresponds to the nature of the build type. Could be “debug” or “release” based on the need.

- `monitor_speed`

  Choose the appropriate baud rate of your serial port. Here we have chosen 115200.

- `monitor_port`

  It is used for configuring the port where the monitor is connected.

- `upload_protocol`

  Fill this field the necessary debug device needed for debug. Here we have chosen ft2232.

- `debug_tool`

  It is similar to the upload_protocol and can be modified with the necessary tool used for debug.

- `debug_speed`

  The debug speed can be any value between 1 MHz to 14 MHz.

- `debug_build_flags`

  Used to specify the flags to be added during build/upload.

Below are the snapshots before and after updating the platformio.ini

---

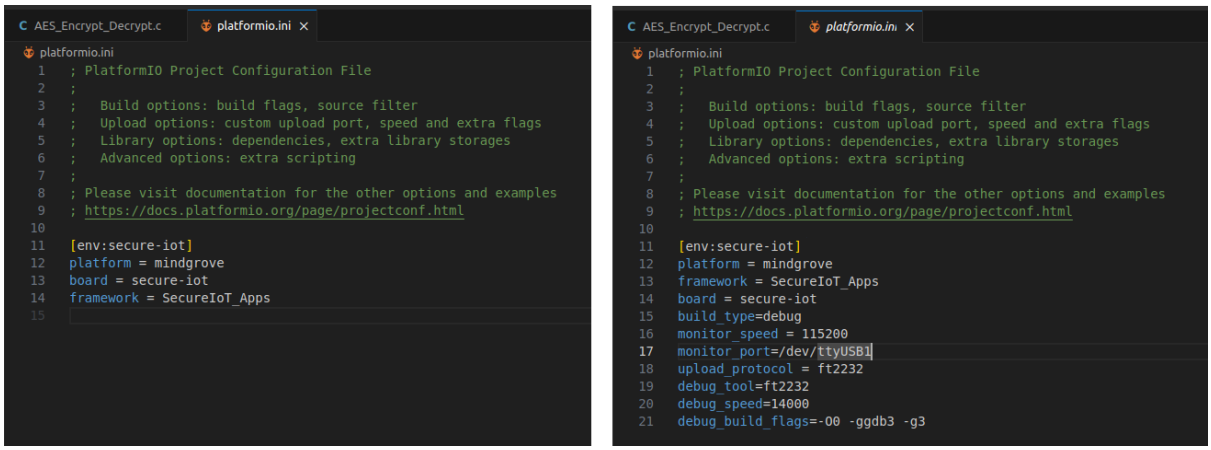


Fig. 5.2.6: PlatformIO configuration file changes

5.2.4 Clean, Build and Upload the application

Options to Clean the repository, Build the Application and Upload(automatically builds and then uploads) the application to the board are present on the Side Bar in PlatformIO Tab. Clicking on Build, Upload, Clean and other options will open the corresponding terminal at the bottom of the screen

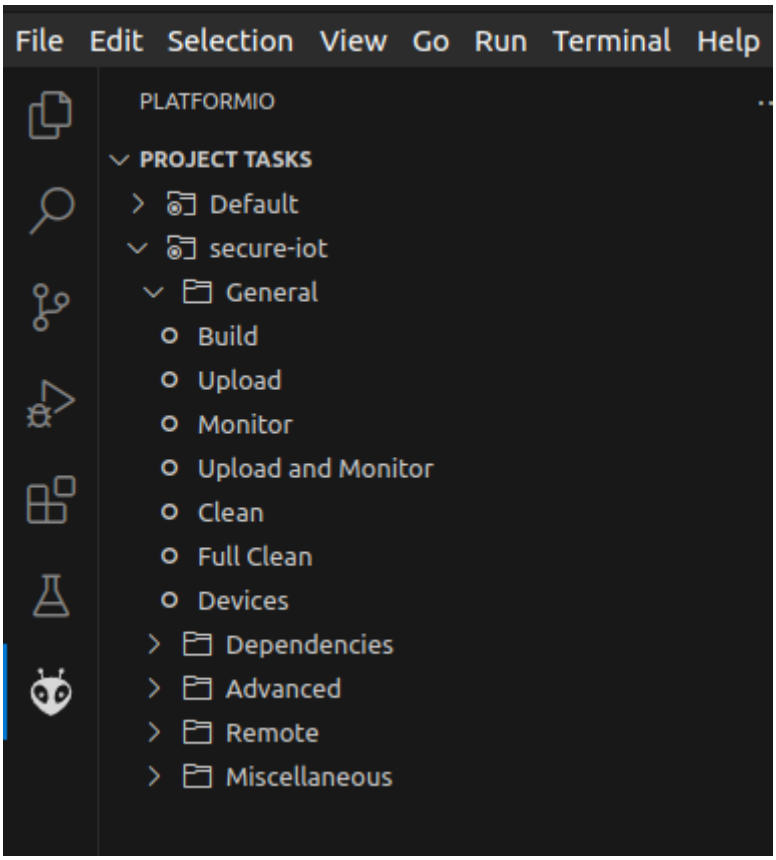


Fig. 5.2.7: Build, clean, and upload operations in PlatformIO

5.2.5 Setting up the serial monitor

To connect to the serial port, choose the Set Monitor icon at the bottom of the IDE and click on the appropriate port.

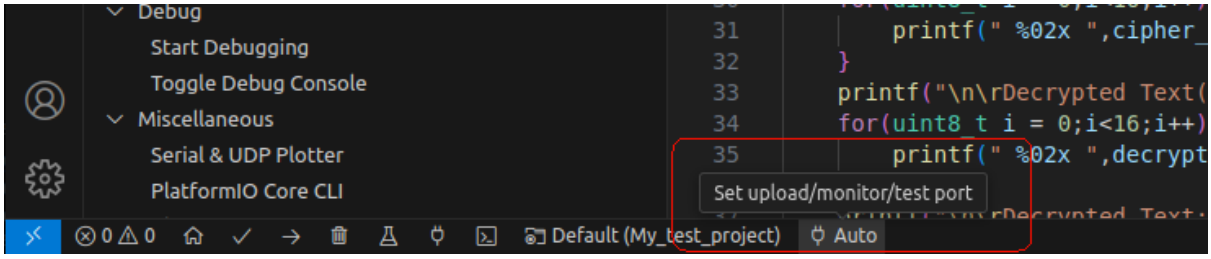


Fig. 5.2.8: Serial monitor setup

5.2.6 Running the application on the board

Clean the workspace and choose the “Upload” option to run the application on the board. At the end of each operation, the result is displayed as success or failure.

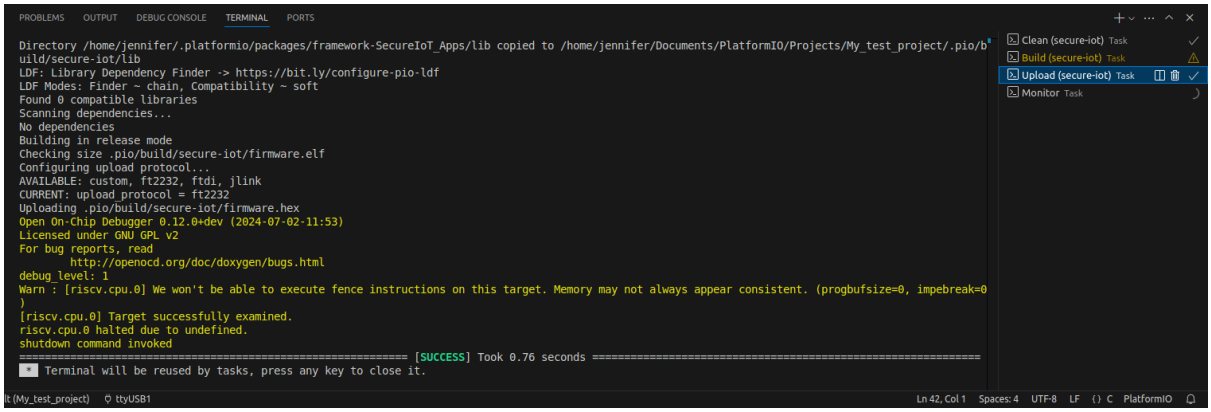


Fig. 5.2.9: Uploading the application to the board

5.2.7 Debugging

For connecting to the debugger, select the Run and Debug Icon on the side bar

During the debug, the options to step through code could be found at the top of the page for quick access. On the left pane, the variables, breakpoints, stacks, registers, etc could be found to support debug.

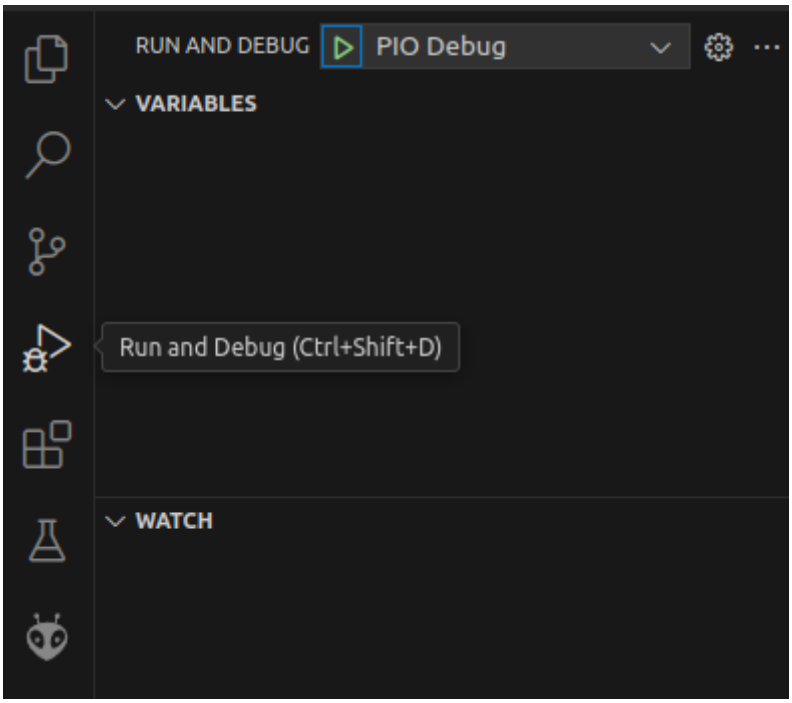


Fig. 5.2.10: Connecting the debugger in PlatformIO

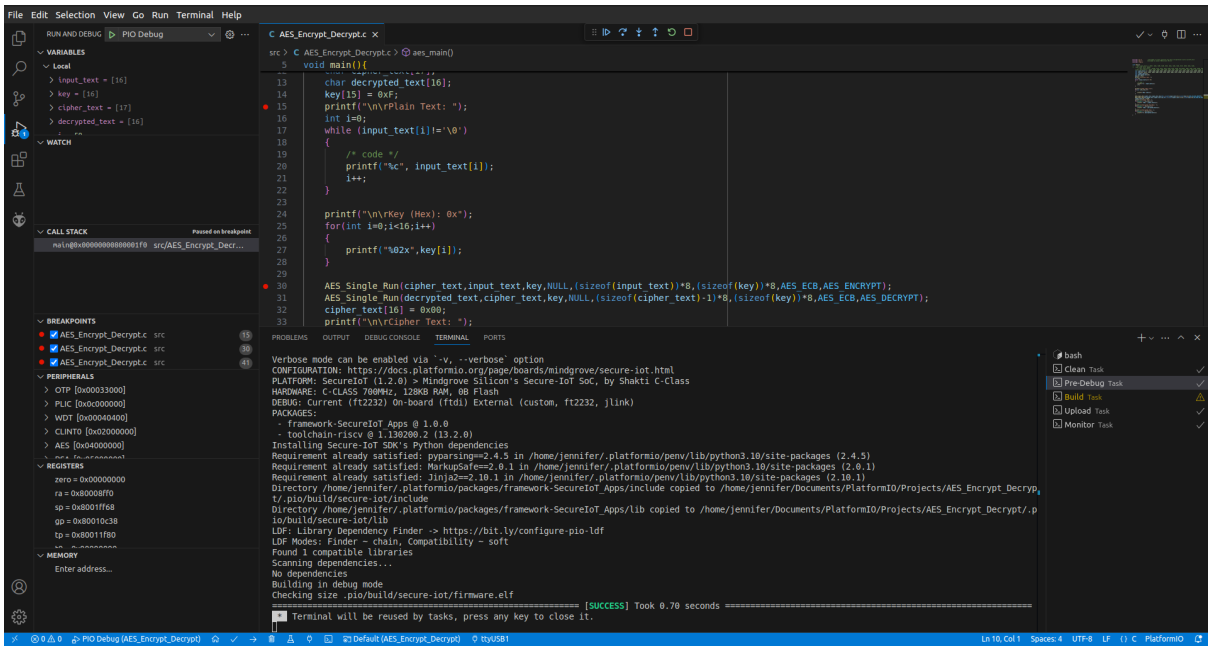


Fig. 5.2.11: Debugger interface features

5.2.8 Output

The Ouput if any to the serial monitor could be seen in the monitor terminal. The printed output of our AES Application could be seen on the serial monitor.

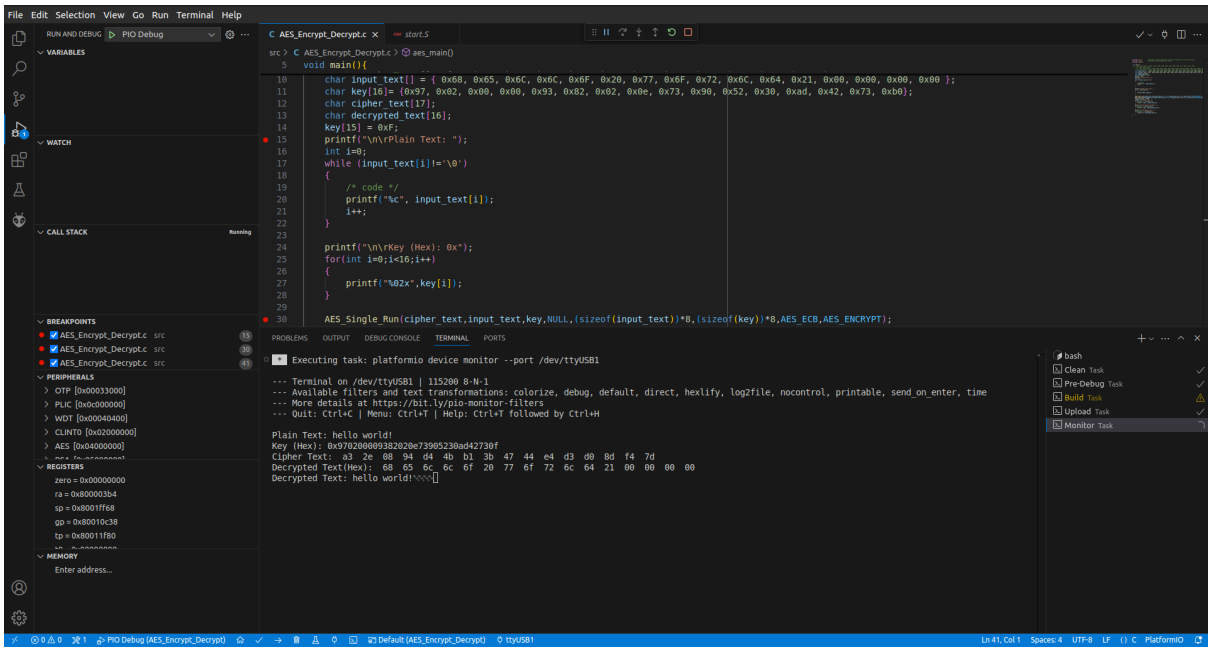


Fig. 5.2.12: Serial monitor output

5.2.9 Troubleshooting

1. **Junk Prints in Serial Monitor**
   - **Possible Causes**: - The Serial port configurations may not be properly configures in platformio.ini.
   - **Solutions**: - Add the correct serial port as well as the baud rate. Reset the board and try again.
2. **Unable to execute any opertaion using PlatformIO IDE - Error : "NotPlatformIOProjectError: Not a PlatformIO project."**
   - **Possible Causes**: - This is because the platformio.ini is not properly configured / the file is missing / the file name is changed.
   - **Solutions**: - Make the necessary changes if incorrectly configured in platformio.ini. - If the file is missing, run the command **"platformio project init"**

from the terminal to create the platformio.ini file within the project. - Once the file is created, add the suitable configurations to upload and debug.

5.3 Makefile

5.3.1 Prerequisites

- A RISC-V toolchain (gcc, g++, etc.) installed in the specified path (default: */opt/riscv*).
- SecureIoT SDK installed, which includes the standard library functions.

5.3.2 Application Organization

Organize your applications in the `src` directory. Each application should be in its own subdirectory. For example:

```
project/
├── src/
│   ├── app1/
│   │   └── main.c
│   └── app2/
│       └── main.cpp
└── Makefile
```

5.3.3 Configuration Options

The Makefile provides several configuration options that can be set via environment variables or directly in the Makefile:

Toolchain

- `RISCV`: Path to the RISC-V toolchain installation. (Default: */opt/riscv*)

Compiler and Linker Flags

- `FLAGS_c`: Flags passed to the C compiler.
- `FLAGS_cpp`: Flags passed to the C++ compiler.
- `MARCH`: CPU architecture (e.g., *rv64imafd_zicsr_zifencei*).
- `MABI`: ABI (e.g., *lp64d*).
- `DEBUG`: Debug build flag. (Set to *y* for debug builds.)

Linker Scripts

`LINKER_SCRIPT`: Path to the linker script. Two options are provided:

- `link.ld` for RAM-based execution.
- `flash-link.ld` for flash-based execution.

Build Targets

Compiling

```
make <app_name> COMPILE
```

Compiles the application and generates the following outputs:

- `work_dir/_tmp_<app>_output/$(PROGRAM).elf`: The compiled ELF file.
- `work_dir/_tmp_<app>_output/$(PROGRAM).dump`: Disassembled binary.
- `work_dir/_tmp_<app>_output/$(PROGRAM).sym`: Symbol table.
- `work_dir/_tmp_<app>_output/code.mem`: Memory image for debugging.
- `work_dir/_tmp_<app>_output/code.bin`: Binary image for flashing.

Running

```
make <app_name> RUN
```

Compiles and runs the application using OpenOCD.

Flashing

```
make <app_name> FLASH
```

Compiles and flashes the application onto the SecureIoT microcontroller using OpenOCD.

Build Output

Compiled artifacts and intermediate files are stored in:

- `work_dir/_tmp_<app>_output/`: Temporary output directory for each application.
- `openocd_uploader/`: Directory for OpenOCD-related files.

Notes

- The SecureIoT SDK provides a standard library in `libsecureiot.a`, which is linked automatically when compiling.
- The build system supports both C and C++ applications.
- The `ALL_applns` target can be used to compile all applications in one go:

```
make ALL_applns
```

Chapter 6. Peripherals

6.1 Analog-to-Digital Converter (ADC)

The ADC (Analog-to-Digital Converter) converts real-world analog signals, such as temperature, light, and pressure, into digital values for processing. This section describes the configuration and usage of the S2401 internal ADC, including initialization steps, register settings and common debugging tips. The ADC supports 12-bit, 10-bit, 8-bit and 6-bit resolutions, operates at sampling rates of up to 70 MSPS, and uses a 1.2 V reference voltage.

6.1.1 ADC Instance Details

There is only one instance for ADC.

Table 6.1.1: ADC Instance Map

ADC Instance	Base Address	Interrupt ID
ADC	0x32000	66

6.1.2 ADC Register Map

The table below has information about the registers of the ADC peripheral, along with their addresses and descriptions. All these registers are 16-bit wide.

Table 6.1.2: ADC Register Map

Register Name	Offset	Description
CTRL	0x00	Initializes the ADC for required mode of operation
Reserved	0x02	Reserved for future use

continues on next page

Table 6.1.2 – continued from previous page

Register Name	Offset	Description
OUTPUT_STATUS	0x04	Holds the output data from the ADC conversion

CTRL

This 16-bit Register is used to configure parameters like input channel, resolution, free-run mode, and interrupt enable flags before starting an ADC conversion.

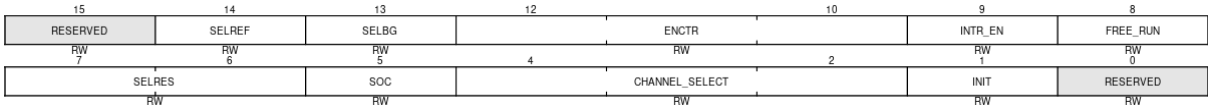


Fig. 6.1.1: ADC Control Register

Table 6.1.3: Bit Field Table

Bits	Field Name	Permission	Description
[0:0]	RESERVED	RW	Reserved for future use; write as 0.
[1:1]	INIT	RW	If set, this bit initializes the ADC
[4:2]	CHAN-NEL_SELECT	RW	Selects the input channel (0-7).
[5:5]	SOC	RW	Triggers the start of conversion.
[7:6]	SELRES	RW	Configures the ADC resolution: • **00**: 6-bit. • **01**: 8-bit. • **10**: 10-bit. • **11**: 12-bit.
[8:8]	FREE_RUN	RW	Enables continuous ADC conversion (free-run mode).
[9:9]	INTR_EN	RW	Enables an interrupt upon conversion completion.

continues on next page

Table 6.1.3 – continued from previous page

Bits	Field Name	Permission	Description
[12:10]	ENCTR	RW	Encoder control or additional ADC configuration.
[13:13]	SELBG	RW	Selects internal bandgap reference for ADC.
[14:14]	SELREF	RO	Selects ADC reference voltage source.
[15:15]	RESERVED	RW	Reserved for future use; write as 0.

Note

The SOC bit should be set only after START_UP_DONE bit is set in OUTPUT register. The hardware sets this START_UP_DONE bit.

OUTPUT_STATUS

The ADC OUTPUT_STATUS Register holds the result of the ADC conversion and the status of conversion.

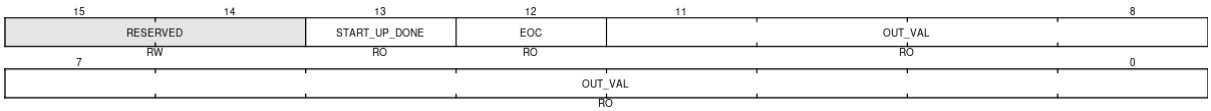


Fig. 6.1.2: ADC OUTPUT Register

Table 6.1.4: Register Bit Fields

Bits	Field Name	Permission	Description
[11:0]	OUT_VAL	RO	Represents the digital value obtained after the ADC conversion.
[12:12]	EOC	RO	Indicates end of conversion.

continues on next page

Table 6.1.4 – continued from previous page

Bits	Field Name	Permission	Description
[13:13]	START_UP.	RO	Set by hardware when ADC initialization is done. Software must wait for this bit before setting SOC feild in CCR register
[15:14]	RESERVED	RW	Reserved for future use; write as 0.

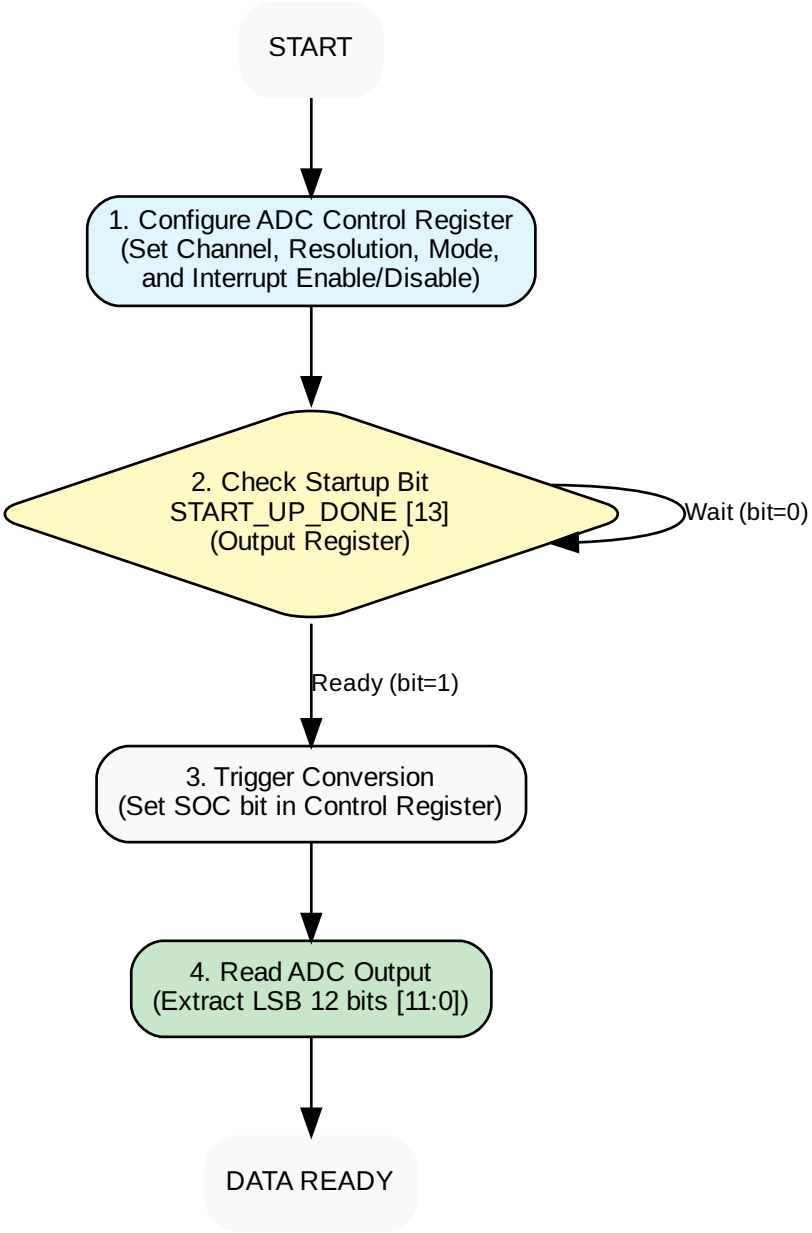
Note

ADC conversion results are MSB-aligned in the 12-bit `OUT_VAL` field for all supported resolutions.

- **6-bit** result → stored in bits `[11:6]`
- **8-bit** result → stored in bits `[11:4]`
- **10-bit** result → stored in bits `[11:2]`
- **12-bit** result → stored in bits `[11:0]`

6.1.3 Workflow

1. **Control Setup:** Configure the ADC Control Register to select the input channel, resolution, interrupt configuration, and operating mode.
2. **Startup Completion Check:** Wait until bit **[13]** (`START_UP_DONE`) in the Output Register is set, indicating that the ADC startup has completed.
3. **Start Conversion:** Set the **SOC** (Start of Conversion) bit in the Control Register to begin sampling.
4. **Data Read:** Read the digital conversion result from bits **[11:0]** of the Output Register.


START
1. Configure ADC Control Register
(Set Channel, Resolution, Mode,
and Interrupt Enable/Disable)
2. Check Startup Bit
START_UP_DONE [13]
(Output Register)
Wait (bit=0)
Ready (bit=1)
3. Trigger Conversion
(Set SOC bit in Control Register)
4. Read ADC Output
(Extract LSB 12 bits [11:0])
DATA READY

6.2 Core Local Interrupt (CLINT)

The core will raise an interrupt to when the *mtime* meets the *mtimecmp* value. This interrupt is known as *Machine Timer Interrupt*.

6.2.1 CLINT Instance Details

Table 6.2.1: Clint Instance Details

Instance	Base Address	Interrupt ID
CLINT	0x2000000	NA

6.2.2 CLINT Register Map

The register map for the CLINT control register.

Table 6.2.2: CLINT Register Mapping for All Configuration Registers

Register Name	Offset	Length (Bits)	Description
msip (Machine Software Interrupt Pending)	0x0000	32	This register generates machine mode software interrupts when set.
mtimecmp (Machine Time Compare)	0x4000	64	This register holds the compare value for the timer.
mtime (Machine Time)	0xBFF8	64	Provides the current timer value.

Machine Software Interrupt Pending Register (MSIP)

The `msip` register is a 32-bit memory-mapped WARL (Write-Any-Read_Legal) register used to control machine-mode software interrupts. Bits [31:1] are hardwired to zero. Bit [0] drives the **MSIP** bit in the `mip` CSR of the corresponding RISC-V hart. Writing a value of 1 to bit [0] sets the software interrupt pending, while writing 0 clears it. On reset, the `msip` register is cleared to zero. For more information on **MSIP**, refer to *Machine Interrupt Registers (mip and mie)*.

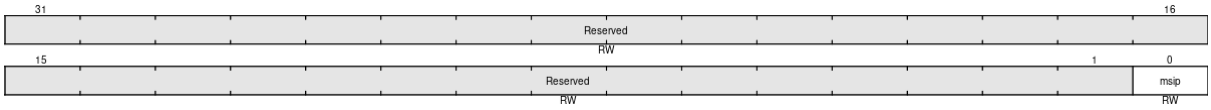


Fig. 6.2.1: msip (Machine Software Interrupt Pending Register)

Table 6.2.3: msip (Machine Software Interrupt Pending Register)

Bits	Field Name	Permission	Description
[0:0]	msip	RW	When set, generates a machine-mode software interrupt. When cleared, no software interrupt is generated.
[31:1]	Reserved	RW	Reserved bits.

Machine Time Compare Register (MTIMECMP)

The `mtimecmp` register holds a 64-bit compare value used by the machine timer. A machine timer interrupt is asserted when `mtime` is greater than or equal to `mtimecmp`, setting the **MTIP** bit in the `mip` CSR. The interrupt remains pending until `mtimecmp` is written with a value greater than the current `mtime`. For more information on **MTIP**, refer to *Machine Interrupt Registers (mip and mie)*.

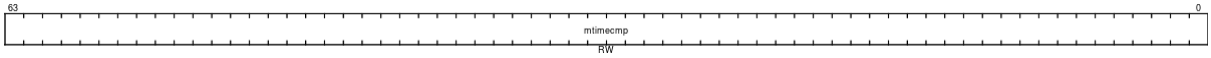


Fig. 6.2.2: mtimecmp (Machine Time Compare Register)

Machine Time Register (MTIME)

The `mtime` register holds a 64-bit monotonically increasing timer value that increments at a constant frequency. The value of `mtime` is continuously compared against `mtimecmp` to determine timer interrupt conditions. A machine timer interrupt is asserted when `mtime` is greater than or equal to `mtimecmp`, which sets the **MTIP** bit in the `mip` CSR. For more information on **MTIP**, refer to *Machine Interrupt Registers (mip and mie)*.

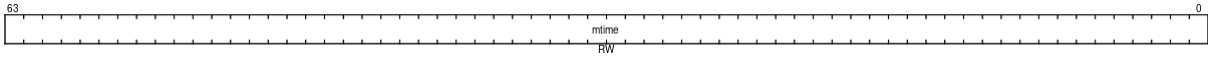
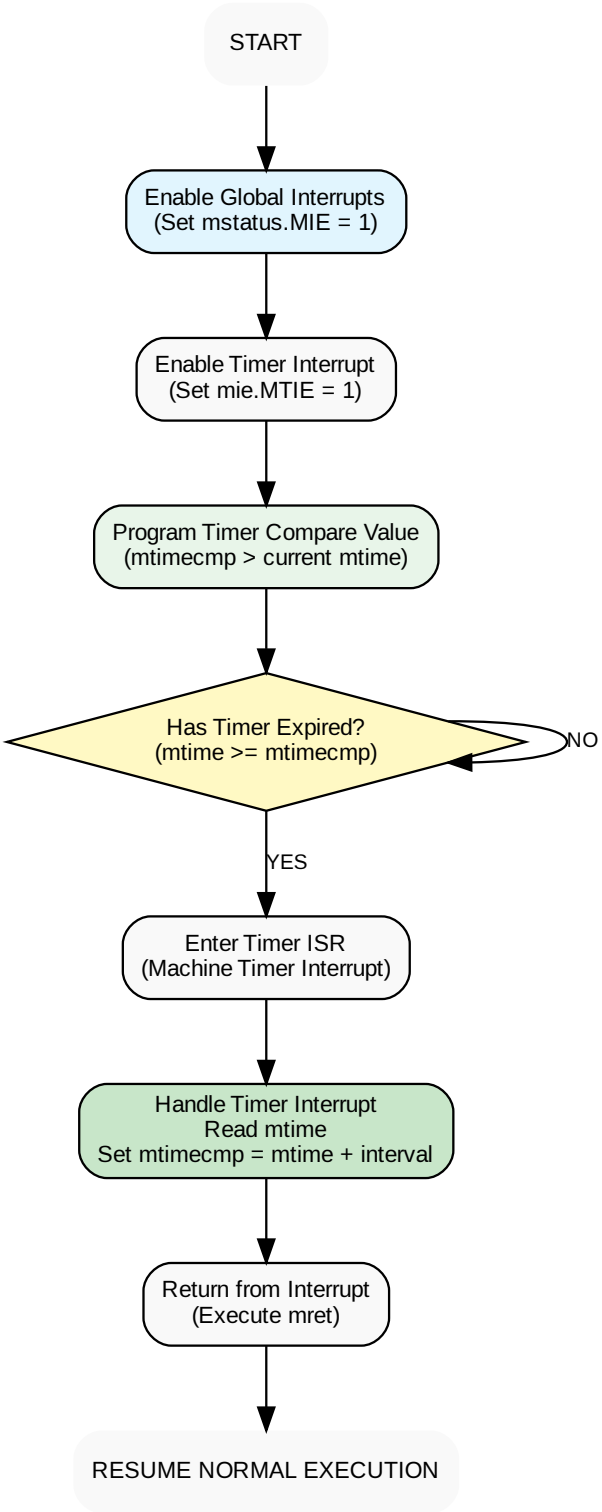


Fig. 6.2.3: mtime (Machine Time Register)

Workflow

1. Enable global interrupts by setting `mstatus.MIE = 1`.
2. Enable machine timer interrupt by setting `mie.MTIE = 1`.
3. Program `mtimecmp` with a value greater than the current `mtime`.
4. Timer runs until `mtime >= mtimecmp`.
5. CPU enters the machine timer interrupt handler when `mtime` reaches or exceeds `mtimecmp`.
6. In the ISR, update `mtimecmp = mtime + interval`
7. Return from interrupt using `mret`.


START
Enable Global Interrupts
(Set mstatus.MIE = 1)
Enable Timer Interrupt
(Set mie.MTIE = 1)
Program Timer Compare Value
(mtimecmp > current mtime)
Has Timer Expired?
(mtime >= mtimecmp)
NO
YES
Enter Timer ISR
(Machine Timer Interrupt)
Handle Timer Interrupt
Read mtime
Set mtimecmp = mtime + interval
Return from Interrupt
(Execute mret)
RESUME NORMAL EXECUTION

6.3 Direct Memory Access (DMA)

Direct Memory Access (DMA) enables peripherals to transfer data to and from memory without relying on the CPU, improving overall system efficiency.

The system provides **multiple DMA channels**, each configurable with its own source, destination, and transfer size.

Channels support the following types of data transfers:

- **Memory-to-Memory** Transfers data directly between two memory locations.
- **Peripheral-to-Memory** Transfers data from a peripheral device to system memory.
- **Memory-to-Peripheral** Transfers data from system memory to a peripheral device.
- **Peripheral-to-Peripheral** Transfers data directly between two peripheral devices.

Each channel includes **interrupt support** for completion and error handling.

6.3.1 DMA Instance Details

Table 6.3.1: DMA Instance Details

Instance	Base Address	Interrupt ID
DMA	0x7000000	68

6.3.2 DMA Register Map

The DMA controller consists of **8 independent channels (Channel 0–7)**. Each DMA channel has the **same logical set of registers**, laid out **back-to-back in memory**. Each channel register block occupies **0x28 bytes** and contains all channel-specific control and address registers. The DMA controller also includes global interrupt registers (`INTERRUPT_STATUS_REG` and `INT_FLAG_CLEAR_REG`) shared across all channels.

Click on a channel entry below to navigate to the **Per-Channel Register Block** description.

Table 6.3.2: DMA Register Map Details

Register Name	Offset	Description
Channel 0 Registers	0x000	Channel 0 register block
Channel 1 Registers	0x028	Channel 1 register block
Channel 2 Registers	0x050	Channel 2 register block
Channel 3 Registers	0x078	Channel 3 register block
Channel 4 Registers	0x0A0	Channel 4 register block
Channel 5 Registers	0x0C8	Channel 5 register block
Channel 6 Registers	0x0F0	Channel 6 register block
Channel 7 Registers	0x118	Channel 7 register block
INTERRUPT_STATUS_REG	0x140	Interrupt Status Register
INT_FLAG_CLEAR_REG	0x148	Interrupt Flag Clear Register

6.3.3 Per-Channel Register Block

Each DMA channel n (where n = 0 to 7) implements the same register layout. The registers belonging to a channel are grouped together as a single block in memory.

This register layout is **common to all DMA channels**

Table 6.3.3: Per-Channel Register Block Details

Register Name	Offset	Length (Bits)	Description
CONFIG_REG	0x00	32	Channel configuration register
TRANSFER_LENGTH_REG	0x08	32	Number of data to transfer
PERIPH_ADDR_REG	0x10	32	Peripheral address register
MEM_ADDR_REG	0x18	32	Memory address register
REQUEST_SELECT_REG	0x20	16	Request select register

DMA Channel Configuration Register (`CONFIG_REG`)

The `CONFIG_REG` register configures the behavior of a DMA channel, including transfer direction, address increment modes, data sizes, priority level, interrupt enables, and transfer mode selection.

This register layout is **common to all DMA channels**. Only the base address differs for each channel.

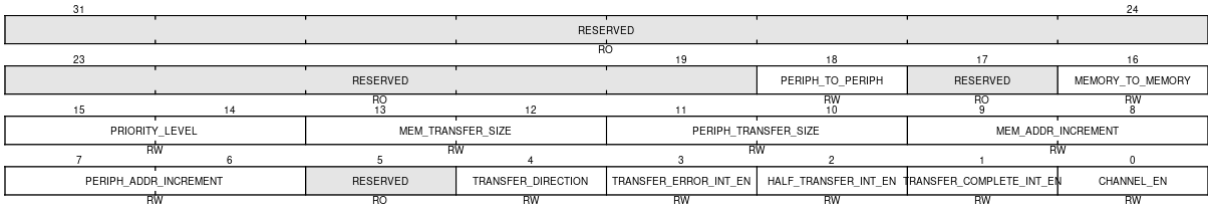


Fig. 6.3.1: DMA Channel Configuration Register

Table 6.3.4: DMA Channel Configuration Register Fields

Bits	Field	Permis-sion	Description
[0:0]	CHANNEL_EN	RW	**Enables or disables the DMA channel.** • **0** : Channel disabled • **1** : Channel enabled Note: **This bit must be cleared before modifying other configuration fields.**
[1:1]	TRANS-FER_COMPLETE_INT_EN	RW	Enables interrupt generation **when the transfer completes.** • **0** : Interrupt disabled • **1** : Interrupt enabled

continues on next page

Table 6.3.4 – continued from previous page

Bits	Field	Permis-sion	Description
[2:2]	HALF_TRANSFER_INT_E	RW	Enables interrupt **when half of the transfer is completed.** • **0** : Interrupt disabled • **1** : Interrupt enabled
[3:3]	TRANS-FER_ERROR_INT_EN	RW	Enables interrupt on **transfer error conditions.** Should be enabled to detect DMA faults • **0** : Interrupt disabled • **1** : Interrupt enabled

continues on next page

Table 6.3.4 – continued from previous page

Bits	Field	Permission	Description
[4:4]	TRANSFER_DIRECTION	RW	Selects the direction of data flow and determines how source and destination attributes are interpreted. **TRANSFER_DIRECTION = 1** **Data is read from memory and written to peripheral (typical M2P)** **Source:** • MEM_ADDR_REG • MEM_TRANSFER_SIZE • PERIPH_ADDR_INCREMENT **Destination:** • PERIPH_ADDR_REG • PERIPH_TRANSFER_SIZE • PERIPH_ADDR_INCREMENT In other modes (like P2P), these same **memory** fields may represent the **source peripheral**, and **peripheral** fields represent the **destination**. **TRANSFER_DIRECTION = 0** **Data is read from peripheral and written to memory (typical P2M)** **Source:** • PERIPH_ADDR_REG • PERIPH_TRANSFER_SIZE • PERIPH_ADDR_INCREMENT **Destination:** • MEM_ADDR_REG • MEM_TRANSFER_SIZE • PERIPH_ADDR_INCREMENT In other modes (like M2M), **these roles may map accordingly based on configuration**, regardless of naming.
[5:5]	RESERVED	RW	Reserved bit.

continues on next page

Table 6.3.4 – continued from previous page

Bits	Field	Permission	Description
[7:6]	PERIPH_ADDR_INCREMEN	RW	Defines the **burst type and address behavior for one side of the transfer**, typically associated with the peripheral register interface. **Encoding:** • **00**: Fixed address (used for fast peripherals) • **01**: Incrementing address (used for memory access) • **10**: Reserved • **11**: Fixed address with support for slow peripherals **Usage:** • Set to **00** for fast peripherals (e.g., SHA, AES, RSA, QSPI) • Set to **11** for slow peripherals (e.g., UART, SPI, ITRACE, ADC, PROIO) • Set to **01** when this field maps to a memory endpoint **Note:** • The actual role (source or destination) of this field depends on the **transfer direction and mode.** • In certain modes (e.g., memory-to-memory or peripheral-to-peripheral), this field may represent **either endpoint regardless of its naming**.

continues on next page

Table 6.3.4 – continued from previous page

Bits	Field	Permission	Description
[9:8]	MEM_ADDR_INCREMEN	RW	Defines the **burst type and address behavior for the other side of the transfer**, typically associated with the memory interface. **Encoding:** • **00**: Fixed address (used for fast peripherals) • **01**: Incrementing address (used for memory access) • **10**: Reserved • **11**: Fixed address with support for slow peripherals **Usage:** • Set to **00** for fast peripherals (e.g., SHA, AES, RSA, QSPI) • Set to **11** for slow peripherals (e.g., UART, SPI, ITRACE, ADC, PROIO) • Set to **01** when this field maps to a memory endpoint **Note:** • Similar to `MEM_ADDR_INCREMENT`, this field does not strictly represent memory in all modes. • Its interpretation depends on **the transfer direction** and mode, and it may correspond to either **source or destination endpoint.**

continues on next page

Table 6.3.4 – continued from previous page

Bits	Field	Permis-sion	Description
[11:10]	PE-RIPH_TRANSFER_SIZE	RW	**Specifies the data width for transfers on the peripheral side.** **Encoding:** • **00**: 8-bit • **01**: 16-bit • **10**: 32-bit • **11**: 64-bit **Note:** • This field defines the access size for the endpoint mapped to the peripheral interface, based on the selected transfer mode and direction.
[13:12]	MEM_TRANSFER_SIZE	RW	**Specifies the data width for transfers on the memory side.** **Encoding:** • **00**: 8-bit • **01**: 16-bit • **10**: 32-bit • **11**: 64-bit **Note:** • This field defines the access size for the endpoint mapped to the memory interface. • In certain modes, this may correspond to either source or destination depending on configuration.

continues on next page

Table 6.3.4 – continued from previous page

Bits	Field	Permission	Description
[15:14]	PRIORITY_LEVEL	RW	**Sets the priority level of the DMA channel.** **Encoding:** • **00**: Low • **01**: Medium • **10**: High • **11**: Very High
[16:16]	MEMORY_TO_MEMORY	RW	Enables memory-to-memory transfer mode. **When set, both source and destination are treated as memory.**
[17:17]	RESERVED	RW	Reserved bit.
[18:18]	PERIPH_TO_PERIPH	RW	Enables peripheral-to-peripheral transfer mode. **When set, both source and destination are treated as peripherals.**
[31:19]	RESERVED	RW	Reserved bits.

Table 6.3.5: DMA Endpoint Mapping and Register Behavior

Mode	TRANSFER_DIRECTIC	Source (Address / Control)	Destination (Address / Control)
Peripheral to Peripheral (P2P)	0	PERIPH_ADDR_REG PERIPH_TRANSFER_SIZE PERIPH_ADDR_INCREMENT	MEM_ADDR_REG MEM_TRANSFER_SIZE MEM_ADDR_INCREMENT
Memory to Peripheral (M2P)	1	MEM_ADDR_REG MEM_TRANSFER_SIZE MEM_ADDR_INCREMENT	PERIPH_ADDR_REG PERIPH_TRANSFER_SIZE PERIPH_ADDR_INCREMENT

continues on next page

Table 6.3.5 – continued from previous page

Mode	TRANSFER_DIRECTIC	Source (Address / Control)	Destination (Address / Control)
Peripheral to Memory (P2M)	0	PERIPH_ADDR_REG PERIPH_TRANSFER_SIZE PERIPH_ADDR_INCREMENT	MEM_ADDR_REG MEM_TRANSFER_SIZE MEM_ADDR_INCREMENT
Memory to Memory (M2M)	0	PERIPH_ADDR_REG PERIPH_TRANSFER_SIZE PERIPH_ADDR_INCREMENT	MEM_ADDR_REG MEM_TRANSFER_SIZE MEM_ADDR_INCREMENT

Note

The interpretation of PERIPH_* and MEM_* depends on the transfer mode and direction. These registers do not strictly correspond to peripheral and memory in all cases.

- When DIR = 1: MEM_* fields define the source, PERIPH_* fields define the destination.
- When DIR = 0: PERIPH_* fields define the source, MEM_* fields define the destination.

In modes such as Memory-to-Memory and Peripheral-to-Peripheral, these fields are reused to represent endpoints irrespective of their naming.

Number of Data to Transfer Register (TRANSFER_LENGTH_REG)

The TRANSFER_LENGTH_REG register stores the number of data (in bytes) to be transferred for the selected DMA channel.

This register layout is **common to all DMA channels**. Only the base address differs for each channel.

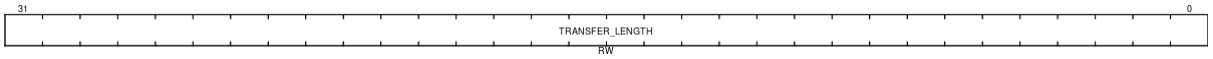


Fig. 6.3.2: Number of Data to Transfer Register (in bytes)

Note

Although the register is 32 bits wide, only the lower 24 bits are used to specify the transfer length.
The maximum value is $2^{24}-1$ (UINT24_MAX). If a value greater than this is written,

the upper 8 bits are ignored and only the lower 24 bits are considered.

Peripheral Address Register (PERIPH_ADDR_REG)

The PERIPH_ADDR_REG register stores the address used as the source or destination during DMA transfers.

This register layout is **common to all DMA channels**. Only the base address differs for each channel.

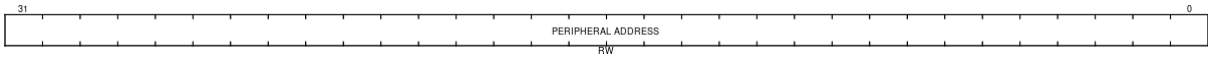


Fig. 6.3.3: Peripheral Address Register

Usage

The interpretation of PERIPH_ADDR_REG depends on the DMA transfer type and the configuration of the TRANSFER_DIRECTION bit.

- **Memory-to-Memory (M2M):**
  - PERIPH_ADDR_REG stores the **source address**.
- **Peripheral-to-Memory (P2M):**
  - PERIPH_ADDR_REG stores the **source address**.
- **Peripheral-to-Peripheral (P2P):**
  - PERIPH_ADDR_REG stores the **source address**.
- **Memory-to-Peripheral (M2P):**
  - PERIPH_ADDR_REG stores the **destination address**.
  - In this mode, the TRANSFER_DIRECTION bit indicates that the peripheral acts as the destination of data.

Note

The TRANSFER_DIRECTION bit determines whether the peripheral side acts as the source or destination in the DMA transfer. Accordingly, the role of PERIPH_ADDR_REG changes between source and destination address.

Memory Address Register (MEM_ADDR_REG)

The MEM_ADDR_REG register stores the memory address used as the source or destination during DMA transfers.

This register layout is **common to all DMA channels**. Only the base address differs for each channel.

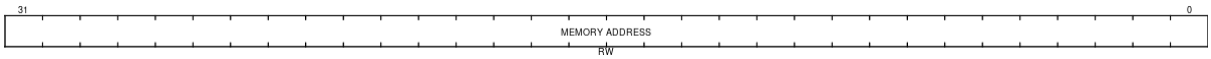


Fig. 6.3.4: Memory Address Register

Usage

The interpretation of MEM_ADDR_REG depends on the DMA transfer type and the configuration of the TRANSFER_DIRECTION bit.

- **Memory-to-Memory (M2M):**
  - MEM_ADDR_REG stores the **destination address**.
- **Peripheral-to-Memory (P2M):**
  - MEM_ADDR_REG stores the **destination address**.
- **Peripheral-to-Peripheral (P2P):**
  - MEM_ADDR_REG stores the **destination address**.
- **Memory-to-Peripheral (M2P):**
  - MEM_ADDR_REG stores the **source address**.
  - In this mode, the TRANSFER_DIRECTION bit indicates that memory acts as the source of data.

Note

The TRANSFER_DIRECTION bit determines whether memory acts as the source or destination in the DMA transfer. Accordingly, the role of MEM_ADDR_REG changes between source and destination address.

Channel Peripheral Request Selection Register (REQUEST_SELECT_REG)

The REQUEST_SELECT_REG register selects the peripheral request signals routed to DMA channel *x* (*x* = 0 to 7). By programming this register, software specifies which peripheral events trigger DMA transfers for the selected channel.

Each DMA channel implements the same REQUEST_SELECT_REG register layout. Only the base address of the register differs per channel.

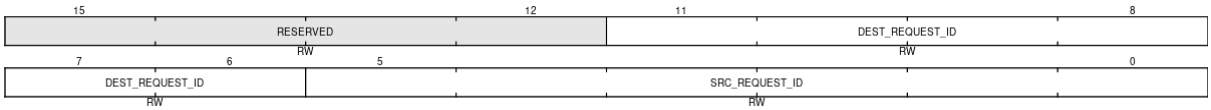


Fig. 6.3.5: Channel Peripheral Request Selection Register

Table 6.3.6: Channel Peripheral Request Selection Register Fields

Bits	Field Name	Permission	Description
[5:0]	SRC_REQUEST_ID	RW	Selects the source peripheral request line for DMA channel *x*. The value corresponds to a request ID defined in *DMA Peripheral Request Mapping*.
[11:6]	DEST_REQUEST_ID	RW	Selects the destination peripheral request line for DMA channel *x*. The value corresponds to a request ID defined in *DMA Peripheral Request Mapping*.
[15:12]	RESERVED	RW	Reserved.

Peripheral Request Encoding

Each value programmed into SRC_REQUEST_ID or DEST_REQUEST_ID corresponds to a fixed peripheral request signal inside the DMA controller.

The mapping between request IDs and peripheral signals is device-specific and is defined in *DMA Peripheral Request Mapping*.

Software must program the appropriate request ID based on the source and destination peripherals used in the DMA transfer.

DMA Peripheral Request Mapping

The table below lists the peripheral request identifiers used by the `REQUEST_SELECT_REG` register. Each identifier corresponds to a fixed peripheral request signal inside the DMA controller.

Table 6.3.7: DMA Peripheral Request IDs

Request Signal Name	Request ID	Description
SHA_OUTP_READY	0	SHA output data ready signal
SHA_CAN_TAKE_INPUT	1	SHA input ready to accept data
RSA_OUTP_READY	2	RSA output data ready signal
RSA_CAN_TAKE_INPUT	3	RSA input ready to accept data
AES_OUTP_READY	4	AES output data ready signal
AES_CAN_TAKE_INPUT	5	AES input ready to accept data
UART0_OUTP_READY	6	UART0 receive data ready signal
UART0_CAN_TAKE_INPUT	7	UART0 transmit ready signal
UART1_OUTP_READY	8	UART1 receive data ready signal
UART1_CAN_TAKE_INPUT	9	UART1 transmit ready signal
UART2_OUTP_READY	10	UART2 receive data ready signal
UART2_CAN_TAKE_INPUT	11	UART2 transmit ready signal
UART3_OUTP_READY	12	UART3 receive data ready signal
UART3_CAN_TAKE_INPUT	13	UART3 transmit ready signal
UART4_OUTP_READY	14	UART4 receive data ready signal
UART4_CAN_TAKE_INPUT	15	UART4 transmit ready signal
SPI0_OUTP_READY	16	SPI0 receive data ready signal
SPI0_CAN_TAKE_INPUT	17	SPI0 transmit ready signal
SPI1_OUTP_READY	18	SPI1 receive data ready signal
SPI1_CAN_TAKE_INPUT	19	SPI1 transmit ready signal

continues on next page

Table 6.3.7 – continued from previous page

Request Signal Name	Request ID	Description
SPI2_OUTP_READY	20	SPI2 receive data ready signal
SPI2_CAN_TAKE_INPUT	21	SPI2 transmit ready signal
SPI3_OUTP_READY	22	SPI3 receive data ready signal
SPI3_CAN_TAKE_INPUT	23	SPI3 transmit ready signal
QSPI1_READY	24	QSPI1 data ready signal
QSPI0_READY	25	QSPI0 data ready signal
PRO_IO_FUSION_OUTP_READ'	26	Pro_IO Fusion (12-bit) output ready
PRO_IO_OCTA_OUTP_READY	27	Pro_IO Octa (8-bit) output ready
PRO_IO_TETRA_OUTP_READY	28	Pro_IO Tetra (4-bit) output ready
PRO_IO_DUO_OUTP_READY	29	Pro_IO Duo (2-bit) output ready
PRO_IO_FUSION_CAN_TAKE_I	30	Pro_IO Fusion (12-bit) input request
PRO_IO_OCTA_CAN_TAKE_INF	31	Pro_IO Octa (8-bit) input request
PRO_IO_TETRA_CAN_TAKE_IN	32	Pro_IO Tetra (4-bit) input request
PRO_IO_DUO_CAN_TAKE_INP	33	Pro_IO Duo (2-bit) input request
ITRACE_OUTP_READY	34	Instruction trace output data ready
ADC_OUTP_READY	35	ADC conversion data ready

Request ID Usage Based on Transfer Type

The usage of `SRC_REQUEST_ID` and `DEST_REQUEST_ID` depends on the configured DMA transfer type.

- **Memory-to-Memory (M2M):**

  When transferring data between memory regions (e.g., RAM, Flash, PSRAM), no peripheral request signals are involved.

  - `SRC_REQUEST_ID` and `DEST_REQUEST_ID` are ignored.
  - Software does not need to program these fields.
- **Peripheral-to-Peripheral (P2P):**

When transferring data between two peripherals:

- SRC_REQUEST_ID must be programmed with the request ID of the **source peripheral**.
- DEST_REQUEST_ID must be programmed with the request ID of the **destination peripheral**.

- **Memory-to-Peripheral (M2P) / Peripheral-to-Memory (P2M):**

  When one side of the transfer is memory and the other is a peripheral:

  - Only SRC_REQUEST_ID needs to be programmed.
  - SRC_REQUEST_ID should contain the request ID of the **peripheral involved in the transfer**, regardless of whether it is acting as source or destination.
  - DEST_REQUEST_ID is ignored.

Usage Examples

The following examples illustrate how to program SRC_REQUEST_ID and DEST_REQUEST_ID based on the DMA transfer type.

Example 1: UART RX to Memory (P2M)

- SRC_REQUEST_ID = UARTx_OUTP_READY
- DEST_REQUEST_ID is ignored

Example 2: Memory to SPI TX (M2P)

- SRC_REQUEST_ID = SPIx_CAN_TAKE_INPUT
- DEST_REQUEST_ID is ignored

Example 3: SPI RX to SPI TX (P2P)

- SRC_REQUEST_ID = SPIx_OUTP_READY
- DEST_REQUEST_ID = SPIy_CAN_TAKE_INPUT

Example 4: Memory to Memory (M2M)

- SRC_REQUEST_ID is ignored
- DEST_REQUEST_ID is ignored

6.3.4 Interrupt Status Register (INTERRUPT_STATUS_REG)

The INTERRUPT_STATUS_REG register provides the status of interrupt flags for all DMA channels, including transfer complete, half-transfer, transfer error, and global interrupt

status bits. Each DMA channel contributes **four interrupt status bits**, and the status for all **eight channels is packed into a single 32-bit register (8 × 4 = 32 bits)**.

This is a global register shared across all channels.

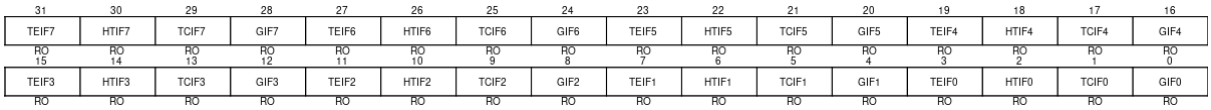


Fig. 6.3.6: Interrupt Status Register

Note

- GIFx = GLOBAL_INT_FLAG_CHx
- TCIFx = TRANSFER_COMPLETE_INT_FLAG_CHx
- HTIFx = HALF_TRANSFER_INT_FLAG_CHx
- TEIFx = TRANSFER_ERROR_INT_FLAG_CHx
  Where x = channel number (0 to 7)

Table 6.3.8: Interrupt Status Register Fields

Bits	Variable Name	Permission	Description
[0:0]	GLOBAL_INT_FLAG_CH0	RO	Global interrupt flag for channel0
[1:1]	TRANSFER_COMPLETE_INT_FLAG_CH	RO	Transfer completion flag for channel0
[2:2]	HALF_TRANSFER_INT_FLAG_CH0	RO	Half transfer flag for channel0
[3:3]	TRANSFER_ERROR_INT_FLAG_CH0	RO	Transfer error flag for channel0
[4:4]	GLOBAL_INT_FLAG_CH1	RO	Global interrupt flag for channel1
[5:5]	TRANSFER_COMPLETE_INT_FLAG_CH	RO	Transfer completion flag for channel1
[6:6]	HALF_TRANSFER_INT_FLAG_CH1	RO	Half transfer flag for channel1

continues on next page

Table 6.3.8 – continued from previous page

Bits	Variable Name	Permission	Description
[7:7]	TRANSFER_ERROR_INT_FLAG_CH1	RO	Transfer error flag for channel1
[8:8]	GLOBAL_INT_FLAG_CH2	RO	Global interrupt flag for channel2
[9:9]	TRANSFER_COMPLETE_INT_FLAG_CH	RO	Transfer completion flag for channel2
[10:10]	HALF_TRANSFER_INT_FLAG_CH2	RO	Half transfer flag for channel2
[11:11]	TRANSFER_ERROR_INT_FLAG_CH2	RO	Transfer error flag for channel2
[12:12]	GLOBAL_INT_FLAG_CH3	RO	Global interrupt flag for channel3
[13:13]	TRANSFER_COMPLETE_INT_FLAG_CH	RO	Transfer completion flag for channel3
[14:14]	HALF_TRANSFER_INT_FLAG_CH3	RO	Half transfer flag for channel3
[15:15]	TRANSFER_ERROR_INT_FLAG_CH3	RO	Transfer error flag for channel3
[16:16]	GLOBAL_INT_FLAG_CH4	RO	Global interrupt flag for channel4
[17:17]	TRANSFER_COMPLETE_INT_FLAG_CH	RO	Transfer completion flag for channel4
[18:18]	HALF_TRANSFER_INT_FLAG_CH4	RO	Half transfer flag for channel4
[19:19]	TRANSFER_ERROR_INT_FLAG_CH4	RO	Transfer error flag for channel4
[20:20	GLOBAL_INT_FLAG_CH5	RO	Global interrupt flag for channel5

continues on next page

Table 6.3.8 – continued from previous page

Bits	Variable Name	Permission	Description
[21:21]	TRANSFER_COMPLETE_INT_FLAG_CH	RO	Transfer completion flag for channel5
[22:22]	HALF_TRANSFER_INT_FLAG_CH5	RO	Half transfer flag for channel5
[23:23]	TRANSFER_ERROR_INT_FLAG_CH5	RO	Transfer error flag for channel5
[24:24]	GLOBAL_INT_FLAG_CH6	RO	Global interrupt flag for channel6
[25:25]	TRANSFER_COMPLETE_INT_FLAG_CH	RO	Transfer completion flag for channel6
[26:26]	HALF_TRANSFER_INT_FLAG_CH6	RO	Half transfer flag for channel6
[27:27]	TRANSFER_ERROR_INT_FLAG_CH6	RO	Transfer error flag for channel6
[28:28]	GLOBAL_INT_FLAG_CH7	RO	Global interrupt flag for channel7
[29:29]	TRANSFER_COMPLETE_INT_FLAG_CH	RO	Transfer completion flag for channel7
[30:30]	HALF_TRANSFER_INT_FLAG_CH7	RO	Half transfer flag for channel7
[31:31]	TRANSFER_ERROR_INT_FLAG_CH7	RO	Transfer error flag for channel7

6.3.5 Interrupt Flag Clear Register (`INT_FLAG_CLEAR_REG`)

The `INT_FLAG_CLEAR_REG` is a global register used to clear interrupt status flags for all DMA channels. It serves as the primary interface for acknowledging interrupts generated by the DMA controller.

Functional Description

Writing a 1 to a specific flag bit in this register clears the corresponding status bit in the INTERRUPT_STATUS_REG (Interrupt Status Register).

Note

Manual Software Toggle Required
Unlike standard Write-1-to-Clear (W1C) registers, this register requires explicit software management of the bit state:

1. To clear an interrupt flag, the software must first **set** the corresponding bit in INT_FLAG_CLEAR_REG to 1.
2. After the interrupt flag in INTERRUPT_STATUS_REG has been cleared, the software **must manually clear** the bit in INT_FLAG_CLEAR_REG by writing a 0 to it.

Failure to write the bit back to 0 may prevent subsequent interrupts from being captured or cleared correctly.

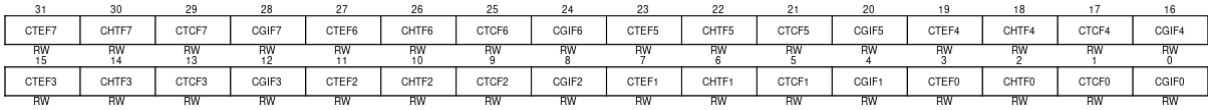


Fig. 6.3.7: Interrupt Flag Clear Register

Note

- CGIFx = CLEAR_GLOBAL_FLAG_CHx
- CTCFx = CLEAR_TRANSFER_COMPLETE_FLAG_CHx
- CHTFx = CLEAR_HALF_TRANSFER_FLAG_CHx
- CTEFx = CLEAR_TRANSFER_ERROR_FLAG_CHx
  Where x = channel number (0 to 7)

Table 6.3.9: Interrupt Flag Clear Register Fields

Bits	Variable Name	Permission	Description
[0:0]	CLEAR_GLOBAL_FLAG_CH0	RW	Global interrupt flag clear for channel0

continues on next page

Table 6.3.9 – continued from previous page

Bits	Variable Name	Permis-sion	Description
[1:1]	CLEAR_TRANSFER_COMPLETE_FLAG_	RW	Transfer completion flag clear for channel0
[2:2]	CLEAR_HALF_TRANSFER_FLAG_CH0	RW	Half transfer flag clear for channel0
[3:3]	CLEAR_TRANSFER_ERROR_FLAG_CH(	RW	Transfer error flag clear for channel0
[4:4]	CLEAR_GLOBAL_FLAG_CH1	RW	Global interrupt flag clear for channel1
[5:5]	CLEAR_TRANSFER_COMPLETE_FLAG_	RW	Transfer completion flag clear for channel1
[6:6]	CLEAR_HALF_TRANSFER_FLAG_CH1	RW	Half transfer flag clear for channel1
[7:7]	CLEAR_TRANSFER_ERROR_FLAG_CH1	RW	Transfer error flag clear for channel1
[8:8]	CLEAR_GLOBAL_FLAG_CH2	RW	Global interrupt flag clear for channel2
[9:9]	CLEAR_TRANSFER_COMPLETE_FLAG_	RW	Transfer completion flag clear for channel2
[10:10]	CLEAR_HALF_TRANSFER_FLAG_CH2	RW	Half transfer flag clear for channel2
[11:11]	CLEAR_TRANSFER_ERROR_FLAG_CH2	RW	Transfer error flag clear for channel2
[12:12]	CLEAR_GLOBAL_FLAG_CH3	RW	Global interrupt flag clear for channel3
[13:13]	CLEAR_TRANSFER_COMPLETE_FLAG_	RW	Transfer completion flag clear for channel3
[14:14]	CLEAR_HALF_TRANSFER_FLAG_CH3	RW	Half transfer flag clear for channel3

continues on next page

Table 6.3.9 – continued from previous page

Bits	Variable Name	Permis-sion	Description
[15:15]	CLEAR_TRANSFER_ERROR_FLAG_CH3	RW	Transfer error flag clear for channel3
[16:16]	CLEAR_GLOBAL_FLAG_CH4	RW	Global interrupt flag clear for channel4
[17:17]	CLEAR_TRANSFER_COMPLETE_FLAG_	RW	Transfer completion flag clear for channel4
[18:18]	CLEAR_HALF_TRANSFER_FLAG_CH4	RW	Half transfer flag clear for channel4
[19:19]	CLEAR_TRANSFER_ERROR_FLAG_CH4	RW	Transfer error flag clear for channel4
[20:20	CLEAR_GLOBAL_FLAG_CH5	RW	Global interrupt flag clear for channel5
[21:21]	CLEAR_TRANSFER_COMPLETE_FLAG_	RW	Transfer completion flag clear for channel5
[22:22	CLEAR_HALF_TRANSFER_FLAG_CH5	RW	Half transfer flag clear for channel5
[23:23	CLEAR_TRANSFER_ERROR_FLAG_CH5	RW	Transfer error flag clear for channel5
[24:24	CLEAR_GLOBAL_FLAG_CH6	RW	Global interrupt flag clear for channel6
[25:25	CLEAR_TRANSFER_COMPLETE_FLAG_	RW	Transfer completion flag clear for channel6
[26:26	CLEAR_HALF_TRANSFER_FLAG_CH6	RW	Half transfer flag clear for channel6
[27:27]	CLEAR_TRANSFER_ERROR_FLAG_CH6	RW	Transfer error flag clear for channel6
[28:28	CLEAR_GLOBAL_FLAG_CH7	RW	Global interrupt flag clear for channel7

continues on next page

Table 6.3.9 – continued from previous page

Bits	Variable Name	Permission	Description
[29:29]	CLEAR_TRANSFER_COMPLETE_FLAG_	RW	Transfer completion flag clear for channel7
[30:30]	CLEAR_HALF_TRANSFER_FLAG_CH7	RW	Half transfer flag clear for channel7
[31:31]	CLEAR_TRANSFER_ERROR_FLAG_CH7	RW	Transfer error flag clear for channel7

6.3.6 Workflow

The following sequence describes a hardware-level DMA workflow using the DMA register interface.

1. **Ensure Channel is Disabled**

   Before configuring the DMA channel:

   - Check CONFIG_REG[CHANNEL_ENABLE] == 0
   - Wait until the channel is idle

   Note: Configuration registers must not be modified when the channel is enabled.

2. **Program Transfer Length**

   Configure the number of data units (In Bytes) to transfer:

   - Write transfer length to `TRANSFER_LENGTH_REG`

3. **Configure Source and Destination Addresses**

   Program endpoint addresses:

   - `PERIPH_ADDR_REG` → one endpoint
   - `MEM_ADDR_REG` → other endpoint

   Note: Which register acts as source/destination depends on TRANSFER_DIRECTION.

4. **Configure Request Selection (for Peripheral Transfers)**

   If peripherals are involved:

   - Select request ID(s) corresponding to source/destination

- Program `REQUEST_SELECT_REG` with REQUEST_ID

5. **Determine Transfer Mode**

   Based on source and destination:

   - Peripheral-to-Peripheral → set PERIPH_TO_PERIPH
   - Memory-to-Memory → set MEMORY_TO_MEMORY
   - Memory-to-Peripheral → use TRANSFER_DIRECTION
   - Peripheral-to-Memory → use TRANSFER_DIRECTION

6. **Configure Transfer Direction**

   Set CONFIG_REG[TRANSFER_DIRECTION]:

   - 0 → Read from peripheral, write to memory (P2M)
   - 1 → Read from memory, write to peripheral (M2P)

   Note: This also determines how `PERIPH_ADDR_REG` and `MEM_ADDR_REG` are interpreted.

7. **Configure Address Increment Modes**

   Set:

   - `PERIPH_ADDR_INCREMENT`
   - `PERIPH_ADDR_INCREMENT`

   Guidelines:

   - Memory → Increment mode
   - Fast peripheral → Fixed burst
   - Slow peripheral → Slow burst mode

8. **Configure Data Width**

   Set transfer sizes:

   - `PERIPH_TRANSFER_SIZE`
   - `MEM_TRANSFER_SIZE`

   Typical values: - 8-bit / 16-bit / 32-bit / 64-bit

9. **Configure Priority**

   Set channel priority:

   - CONFIG_REG[PRIORITY_LEVEL]

10. **Configure Interrupts (Optional)**

    Enable required interrupts:

    - TRANSFER_COMPLETE_INT_EN

- HALF_TRANSFER_INT_EN
- TRANSFER_ERROR_INT_EN

11. **Enable DMA Channel**

    Start the transfer:

    - Set CONFIG_REG[CHANNEL_ENABLE] = 1

12. **DMA Executes Transfer**

    DMA performs the transfer autonomously:

    - Reads from source
    - Writes to destination
    - Updates addresses based on increment settings
    - Decrements `TRANSFER_LENGTH_REG`

13. **Monitor Transfer Status**

    Two methods:

    Polling Mode:

    - Read interrupt flags:
      - TRANSFER_COMPLETE_INT_FLAG_CH0
      - HALF_TRANSFER_INT_FLAG_CH0
      - TRANSFER_ERROR_INT_FLAG_CH0
      - GLOBAL_INT_FLAG_CH0

    Interrupt Mode:

    - DMA triggers interrupt when enabled events occur

14. **Handle Completion**

    On transfer completion or error:

    - Process data
    - Take appropriate action based on flags

15. **Disable DMA Channel**

    Stop DMA:

    - Clear CONFIG_REG[CHANNEL_ENABLE]

16. **Clear Interrupt Flags**

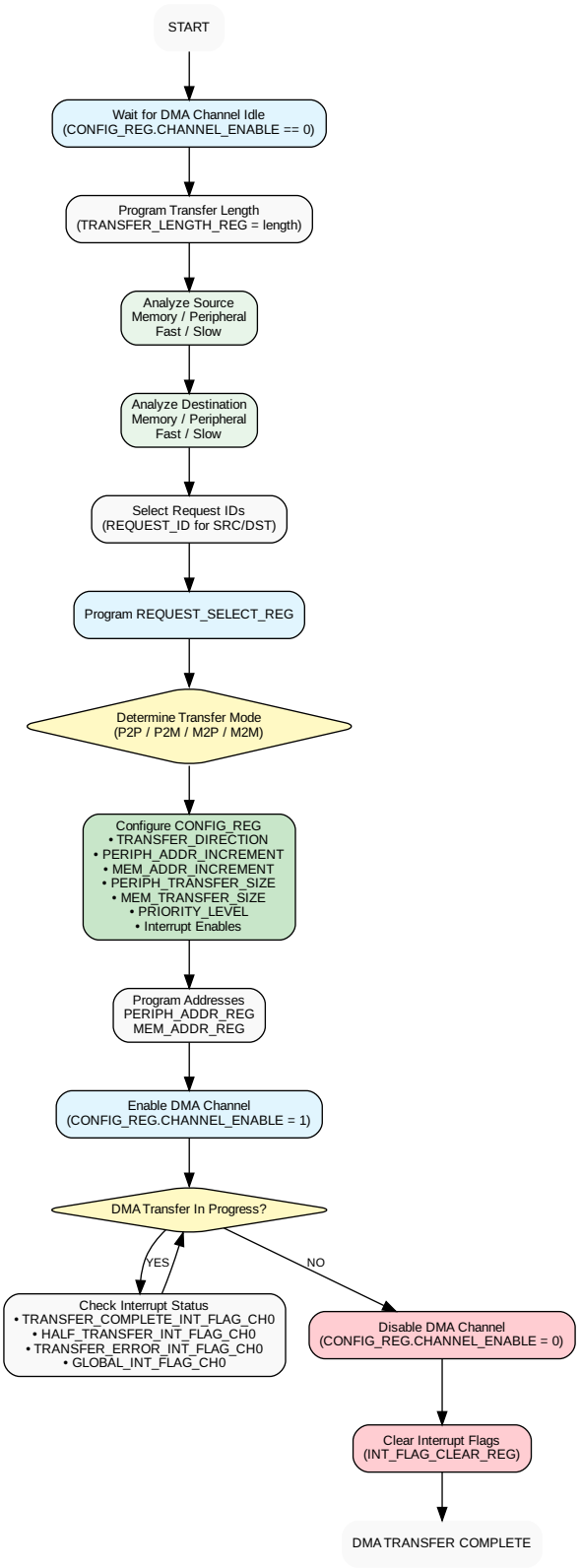
    Clear flags using `INT_FLAG_CLEAR_REG`:

    - CLEAR_GLOBAL_FLAG_CH0
    - CLEAR_TRANSFER_COMPLETE_FLAG_CH0

- CLEAR_HALF_TRANSFER_FLAG_CH0
- CLEAR_TRANSFER_ERROR_FLAG_CH0

17. **Channel Ready for Reuse**

    DMA channel is now ready for next transfer.


START
Wait for DMA Channel Idle
(CONFIG_REG.CHANNEL_ENABLE == 0)
Program Transfer Length
(TRANSFER_LENGTH_REG = length)
Analyze Source
Memory / Peripheral
Fast / Slow
Analyze Destination
Memory / Peripheral
Fast / Slow
Select Request IDs
(REQUEST_ID for SRC/DST)
Program REQUEST_SELECT_REG
Determine Transfer Mode
(P2P / P2M / M2P / M2M)
Configure CONFIG_REG
• TRANSFER_DIRECTION
• PERIPH_ADDR_INCREMENT
• MEM_ADDR_INCREMENT
• PERIPH_TRANSFER_SIZE
• MEM_TRANSFER_SIZE
• PRIORITY_LEVEL
• Interrupt Enables
Program Addresses
PERIPH_ADDR_REG
MEM_ADDR_REG
Enable DMA Channel
(CONFIG_REG.CHANNEL_ENABLE = 1)
DMA Transfer In Progress?
YES
NO
Check Interrupt Status
• TRANSFER_COMPLETE_INT_FLAG_CH0
• HALF_TRANSFER_INT_FLAG_CH0
• TRANSFER_ERROR_INT_FLAG_CH0
• GLOBAL_INT_FLAG_CH0
Disable DMA Channel
(CONFIG_REG.CHANNEL_ENABLE = 0)
Clear Interrupt Flags
(INT_FLAG_CLEAR_REG)
DMA TRANSFER COMPLETE

6.4 General Purpose Input and Output(GPIO)

In S2401, General Purpose Input/Output (GPIO) provide a flexible interface between the processor and external hardware components. GPIO pins can be configured as inputs or outputs to read external signals or drive devices such as LEDs, switches, sensors, and control lines.

In addition to standard GPIO functionality, many of the pin supports Pinmux, allowing it to be dynamically assigned to alternate peripheral functions such as PWM, SPI, UART, GPTIMER, or JTAG. This capability enables multiple internal peripherals to share the same physical pins, offering greater flexibility in system design without requiring hardware layout changes. When a pin is configured for an alternate function via Pinmux, the corresponding GPIO registers no longer control that pin.

Beyond basic digital I/O, S2401 also supports a ProIO feature, which enables high-speed and continuous data transfer by grouping multiple GPIO pins into hardware-managed data buffers. For details on ProIO operation, buffer grouping, and data flow, refer to *ProIO*

Note

For detailed information on GPIO pinmux, refer to *Pinmux Register Map*.

6.4.1 Instance Details

GPIO pins are divided into two register instances based on their pin number range. Each instance has a separate base address and interrupt mapping. Software must select the appropriate base address and interrupt line corresponding to the GPIO pin being accessed.

Table 6.4.1: GPIO Instance Map

GPIO Instance	Base Address	Interrupt ID
GPIO0 - GPIO31	0x00040200	0 - 31
GPIO32 - GPIO44	0x00040300	69 - 81

Note

GPIO interrupt lines follow a fixed one-to-one mapping with GPIO pin numbers. For GPIO pins 0–31, the interrupt number corresponds directly to the GPIO index (for example, GPIO0 → GPIO0_IRQn = 1, GPIO1 → GPIO1_IRQn = 2).
For GPIO pins 32–44, interrupts are mapped to the secondary GPIO interrupt range (GPIOP0_IRQn – GPIOP12_IRQn), where GPIOP0_IRQn corresponds to GPIO32, GPIOP1_IRQn to GPIO33, and so on. (for example, GPIO32 → GPIOP0_IRQn = 69, GPIO33 → GPIOP1_IRQn = 70)

6.4.2 ProIO

The S2401 supports a ProIO feature that enables continuous, high-speed data transfer through GPIO pins by grouping multiple pins into fixed-width data interfaces. Instead of operating on individual GPIO pins, selected pins are organized into 2-bit, 4-bit, 8-bit, or combined 12-bit data groups.

In addition to individual groups, the module supports a combined 12-bit interface mode. In this mode, the 4-bit (Tetra) group and the 8-bit (Octa) group operate together as a single logical data path:

- The 4-bit group provides the Most Significant Bits (MSBs)
- The 8-bit group provides the Least Significant Bits (LSBs)

This configuration is intended for peripherals or interfaces that require a 12-bit data width, enabling seamless parallel data capture or transmission using GPIO pins.

Each ProIO group includes an independent FIFO with a fixed storage capacity of 32 bits. While the FIFO depth is internally fixed, the number of usable entries depends on the data width:

- A 2-bit group stores 16 entries, each 2 bits wide (16 × 2 bits = 32 bits total)
- A 4-bit group stores 8 entries, each 4 bits wide (8 × 4 bits = 32 bits total)
- A 8-bit group stores 4 entries, each 8 bits wide (4 × 8 bits = 32 bits total)

This design ensures that each ProIO group always stores exactly 32 bits of data, regardless of the selected data width.

allowing data to be queued and processed efficiently without software intervention on every GPIO transition.

Each ProIO group operates with its own clock source, which can be selected independently as either an internal clock or an external clock provided through a designated GPIO pin. This flexibility allows the ProIO to interface with a wide range of external

peripherals and data rates.

Table 6.4.2: ProIO Grouping

ProIO Group	Clock GPIO	Data GPIOs
DUO	GPIO 5	GPIO 3, 4
TETRA	GPIO 6	GPIO 11, 15, 18, 22
OCTA	GPIO 7	GPIO 1, 2, 8, 9, 14, 16, 17, 31
FUSION	GPIO 6, 7	GPIO 11, 15, 18, 22, 1, 2, 8, 9, 14, 16, 17, 31

Note

DMA can enqueue or dequeue data only when buffer space or data is available. Maximum throughput is achieved when DMA transfers occur in parallel with continuous GPIO-driven enqueue/dequeue operations.

6.4.3 Register Map and Details

The table below lists the registers of the GPIO peripheral along with their addresses and descriptions.

GPIO pins 0–31 use base address `0x40200`, The correct base address must be selected according to the GPIO pin number.

Table 6.4.3: GPIO Register Map

Register Name	Offset	Length (In Bits)	Description
GPIO_DIRECTION (Direction Register)	0x00	32	Configures the direction of GPIO pins as input (0) or output (1).

continues on next page

Table 6.4.3 – continued from previous page

Register Name	Offset	Length (In Bits)	Description
GPIO_DATA (Data Register)	0x08	32	Reads the logic level of input pins or reflects the output data state.
GPIO_SET (Set Register)	0x10	32	Sets the specified GPIO output pins to HIGH.
GPIO_CLEAR (Clear Register)	0x18	32	Clears the specified GPIO output pins to LOW.
GPIO_TOGGLE (Toggle Register)	0x20	32	Toggles the current state of the specified GPIO output pins.
GPIO_INTR (Interrupt Enable Register)	0x30	32	Enables interrupt generation for the specified GPIO pins.
GPIO_PULLUP_CONFIG (Pull-Up Configuration Register)	0x38	32	Enables or disables internal pull-up resistors for GPIO pins.
PRO_IO_CONTROL (Buffer Control Register)	0x40	32	Controls PRO_IO buffer configuration including enable, direction, clock selection, clock edge selection, and data size checks for DUO, TETRA, OCTA, and FUSION.
PRO_IO_STATUS (Buffer Status Register)	0x48	32	Indicates buffer status including FIFO full/empty conditions and DMA readiness (can take input / output ready) for DUO, TETRA, OCTA, and FUSION.
PRO_IO_DUO_CLOCK_PRESCALER (2-Bit Buffer Clock Prescaler Register)	0x50	32	Configures the internal clock prescaler for the DUO (2-bit) buffer when operating with internal clock.

continues on next page

Table 6.4.3 – continued from previous page

Register Name	Offset	Length (In Bits)	Description
PRO_IO_TETRA_CLOCK_PRESCALER (4-Bit Buffer Clock Prescaler Register)	0x58	32	Configures the internal clock prescaler for the TETRA (4-bit) buffer when operating with internal clock.
PRO_IO_OCTA_CLOCK_PRESCALER (8-Bit Buffer Clock Prescaler Register)	0x60	32	Configures the internal clock prescaler for the OCTA (8-bit) buffer when operating with internal clock.
PRO_IO_FUSION_CLOCK_PRESCALER (12-Bit Buffer Clock Prescaler Register)	0x68	32	Configures the internal clock prescaler for the FUSION (12-bit combined) buffer when operating with internal clock.
PRO_IO_DUO_DATA (2-Bit Buffer Data Register)	0x70	32	Transfers data to or from the DUO (2-bit) buffer based on the configured clock and direction.
PRO_IO_TETRA_DATA (4-Bit Buffer Data Register)	0x78	32	Transfers data to or from the TETRA (4-bit) buffer based on the configured clock and direction.
PRO_IO_OCTA_DATA (8-Bit Buffer Data Register)	0x80	32	Transfers data to or from the OCTA (8-bit) buffer based on the configured clock and direction.
PRO_IO_FUSION_DATA (12-Bit Buffer Data Register)	0x88	32	Transfers combined 12-bit data using the FUSION buffer, composed of TETRA (MSB) and OCTA (LSB) segments.

For GPIO pins 32–44 use base address `0x40300`. The correct base address must be selected according to the GPIO pin number.

Table 6.4.4: GPIO Register Map

Register Name	Offset	Length (In Bits)	Description
GPIO_DIRECTION (Direction Register)	0x00	32	Configures the direction of GPIO pins as input (0) or output (1).
GPIO_DATA (Data Register)	0x08	32	Reads the logic level of input pins or reflects the output data state.
GPIO_SET (Set Register)	0x10	32	Sets the specified GPIO output pins to HIGH.
GPIO_CLEAR (Clear Register)	0x18	32	Clears the specified GPIO output pins to LOW.
GPIO_TOGGLE (Toggle Register)	0x20	32	Toggles the current state of the specified GPIO output pins.
GPIO_INTR (Interrupt Enable Register)	0x30	32	Enables interrupt generation for the specified GPIO pins.
GPIO_PULLUP_CONFIG (Pull-Up Configuration Register)	0x38	32	Enables or disables internal pull-up resistors for GPIO pins.

Direction Register (GPIO_DIRECTION)

The **GPIO_DIRECTION** is a 32-bit register controls the direction of the GPIO pins, with each bit corresponding to a single GPIO pin (Bit 0 controls GPIO0, Bit 1 controls GPIO1, and so on). Writing a value of 0 to a bit configures the associated GPIO pin as an input, allowing external signals to be read, while writing a value of 1 configures the pin as an output, enabling the device to drive the pin.

Fig. 6.4.1: GPIO_DIRECTION

Note

For GPIO pins 0–31, this register must be accessed using the base address 0x40200. For GPIO pins 32–44, the same register applies, but it must be accessed using the base address 0x40300. The appropriate base address must be selected based on the GPIO pin number being configured.

Data Register (GPIO_DATA)

The **GPIO_DATA** is a 32-bit register provides access to the data value of each GPIO pin, with each bit corresponding to a specific GPIO pin. When a GPIO pin is configured as an output, writing to the corresponding bit drives the logic level on the pin. When a GPIO pin is configured as an input, reading the corresponding bit returns the current logic level present on the pin.

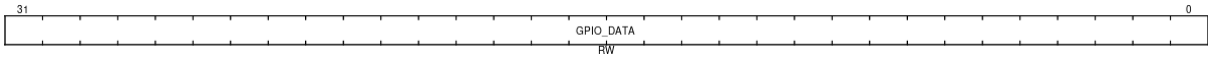


Fig. 6.4.2: GPIO_DATA

Note

For GPIO pins 0–31, this register must be accessed using the base address 0x40200. For GPIO pins 32–44, the same register applies, but it must be accessed using the base address 0x40300. The appropriate base address must be selected based on the GPIO pin number being configured.

Set Register (GPIO_SET)

The **GPIO_SET** is an 32-bit write-only register, used to set selected GPIO pins to a logic high level without affecting the state of other pins. Each bit corresponds to a specific GPIO pin. Writing a value of 1 to a bit drives the corresponding GPIO pin high, provided the pin is configured as an output. Writing 0 has no effect on the pin state.

Fig. 6.4.3: GPIO_SET

Note

For GPIO pins 0–31, this register must be accessed using the base address 0x40200. For GPIO pins 32–44, the same register applies, but it must be accessed using the base address 0x40300. The appropriate base address must be selected based on the GPIO pin number being configured.

Clear Register (GPIO_CLEAR)

The **GPIO_CLEAR** is an 32-bit write-only register, used to clear selected GPIO pins to a logic low level without affecting the state of other pins. Each bit corresponds to a specific GPIO pin. Writing a value of 1 to a bit drives the corresponding GPIO pin low, provided the pin is configured as an output. Writing 0 has no effect on the pin state.

Fig. 6.4.4: GPIO_CLEAR

Note

For GPIO pins 0–31, this register must be accessed using the base address 0x40200. For GPIO pins 32–44, the same register applies, but it must be accessed using the base address 0x40300. The appropriate base address must be selected based on the GPIO pin number being configured.

Toggle Register (GPIO_TOGGLE)

The **GPIO_TOGGLE** is an 32-bit write-only register, used to invert the current logic state of selected GPIO pins without affecting the state of other pins. Each bit corresponds to a specific GPIO pin. Writing a value of 1 to a bit toggles the logic level of the corresponding GPIO pin, provided the pin is configured as an output. Writing 0 has no effect on the pin state.

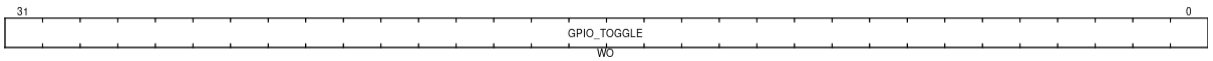


Fig. 6.4.5: GPIO_TOGGLE

Note

For GPIO pins 0–31, this register must be accessed using the base address 0x40200. For GPIO pins 32–44, the same register applies, but it must be accessed using the base address 0x40300. The appropriate base address must be selected based on the GPIO pin number being configured.

Interrupt Enable Register (GPIO_INTR)

The **GPIO_INTR** is a 32-bit register, used to enable interrupt for selected GPIO pins. Each bit corresponds to a specific GPIO pin. When a bit is set to 1, interrupt is enabled for the corresponding GPIO pin. When a bit is cleared to 0, interrupts are disabled for that pin.

Fig. 6.4.6: GPIO_INTR

Note

For GPIO pins 0–31, this register must be accessed using the base address 0x40200. For GPIO pins 32–44, the same register applies, but it must be accessed using the base address 0x40300. The appropriate base address must be selected based on the GPIO pin number being configured.

Pull-Up Configuration Register (GPIO_PULLUP_CONFIG)

The **GPIO_PULLUP_CONFIG** is a 32-bit register controls the internal pull-up resistors for GPIO pins, with each bit corresponding to a specific GPIO pin. The pull-up resistors are applied to the GPIO I/O pads. Writing a value of 1 to a bit enables the internal pull-up resistor for the corresponding GPIO pin. Writing a value of 0 disables the pull-up resistor, leaving the pin in a floating state.

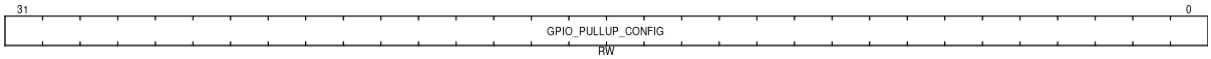


Fig. 6.4.7: GPIO_PULLUP_CONFIG

Note

For GPIO pins 0–31, this register must be accessed using the base address 0x40200. For GPIO pins 32–44, the same register applies, but it must be accessed using the base address 0x40300. The appropriate base address must be selected based on the GPIO pin number being configured.

PRO IO Buffer Control Register (PRO_IO_CONTROL)

The **PRO_IO_CONTROL** is a 32-bit register controls the buffering functionality of the GPIO. It allows enabling or disabling different buffer groups, selecting the clock source and clock edge, configuring data direction, clearing buffer contents, and setting data-availability check thresholds for ProIO operations.

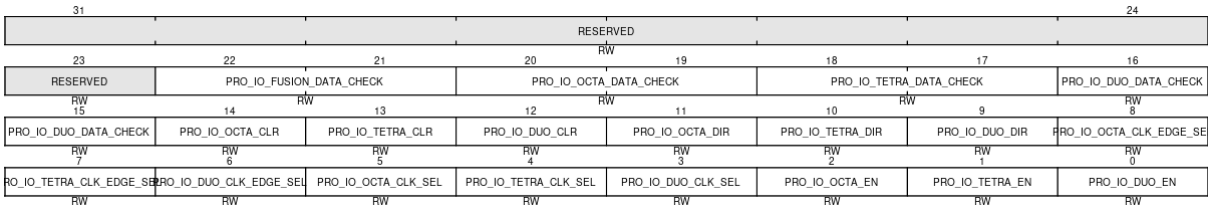


Fig. 6.4.8: PRO_IO_CONTROL

PRO IO Status Register (PRO_IO_STATUS)

The **PRO_IO_STATUS** is a 32-bit register provides the current status of the ProIO subsystem. It provides information about buffer fill levels and DMA readiness for the DUO, TETRA, OCTA, and FUSION groups. Each status bit reflects the availability of buffer space or data for enqueue, dequeue, or DMA transfer operations.

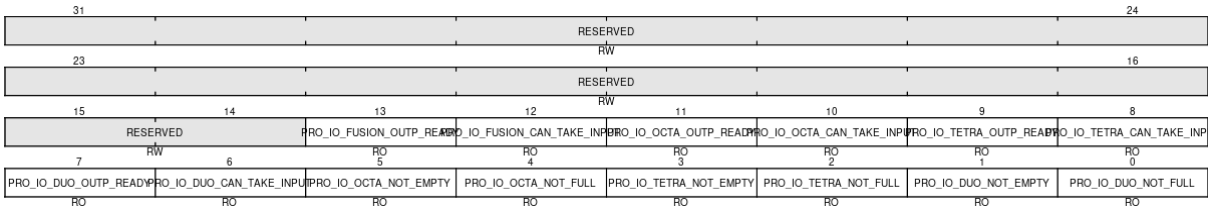


Fig. 6.4.9: PRO_IO_STATUS

PRO IO DUO Clock Prescaler (PRO_IO_DUO_CLOCK_PRESCALER)

The **GPIO_BUFFER_2_CLOCK_PRESCALER** is a 32-bit register configures the clock prescaler for the **internal clock** source of the Buffer-2. The prescaler divides the internal clock frequency to generate the required sampling rate for Buffer-2 operations.

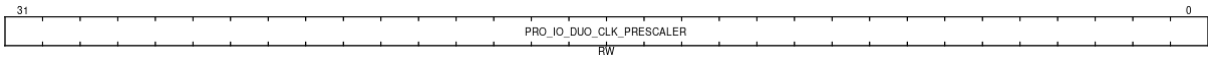


Fig. 6.4.10: PRO_IO_DUO_CLOCK_PRESCALER

PRO IO TETRA Clock Prescaler (PRO_IO_TETRA_CLOCK_PRESCALER)

The **PRO_IO_TETRA_CLOCK_PRESCALER** is a 32-bit register configures the clock prescaler for the **internal clock** source of the Buffer-4. The prescaler divides the internal clock frequency to generate the required sampling rate for Buffer-4 operations.

Fig. 6.4.11: PRO_IO_TETRA_CLOCK_PRESCALER

PRO IO OCTA Clock Prescaler (PRO_IO_OCTA_CLOCK_PRESCALER)

The **PRO_IO_OCTA_CLOCK_PRESCALER** is a 32-bit register configures the clock prescaler for the **internal clock** source of the Buffer-8. The prescaler divides the internal clock frequency to generate the required sampling rate for Buffer-8 operations.

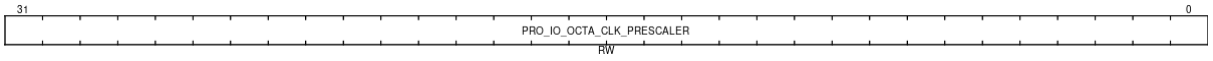


Fig. 6.4.12: PRO_IO_OCTA_CLOCK_PRESCALER

PRO IO FUSION Clock Prescaler (PRO_IO_FUSION_CLOCK_PRESCALER)

The **PRO_IO_FUSION_CLOCK_PRESCALER** is a 32-bit register configures the clock prescaler for the **internal clock** source of the Buffer-4_8. The prescaler divides the internal clock frequency to generate the required sampling rate for Buffer-4_8 operations.

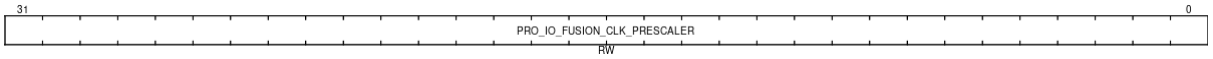


Fig. 6.4.13: PRO_IO_FUSION_CLOCK_PRESCALER

PRO IO DUO Data Register (PRO_IO_DUO_DATA)

The PRO_IO_DUO_DATA is a 32-bit register used for data transactions synchronized to the configured buffer clock. It supports **8-bit**, **16-bit**, and **32-bit** accesses, allowing flexible data transfer widths based on the selected buffer configuration and application requirements.

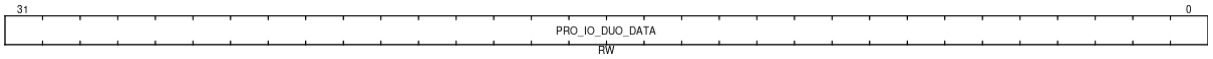


Fig. 6.4.14: PRO_IO_DUO_DATA

PRO IO TETRA Data Register (PRO_IO_TETRA_DATA)

The PRO_IO_TETRA_DATA is a 32-bit register used for data transactions synchronized to the configured buffer clock. It supports **8-bit**, **16-bit**, and **32-bit** accesses, allowing flexible data transfer widths based on the selected buffer configuration and application requirements.

Fig. 6.4.15: PRO_IO_TETRA_DATA

PRO IO OCTA Data Register (PRO_IO_OCTA_DATA)

The PRO_IO_OCTA_DATA is a 32-bit register used for data transactions synchronized to the configured buffer clock. It supports **8-bit**, **16-bit**, and **32-bit** accesses, allowing flexible data transfer widths based on the selected buffer configuration and application requirements.

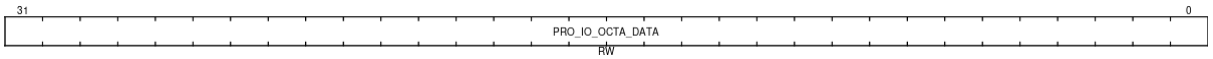


Fig. 6.4.16: PRO_IO_OCTA_DATA

PRO IO FUSION Data Register (PRO_IO_FUSION_DATA)

The PRO_IO_FUSION_DATA is a 32-bit register used for data transactions synchronized to the configured buffer clock. It supports **8-bit**, **16-bit**, and **32-bit** accesses, allowing flexible data transfer widths based on the selected buffer configuration and application requirements.

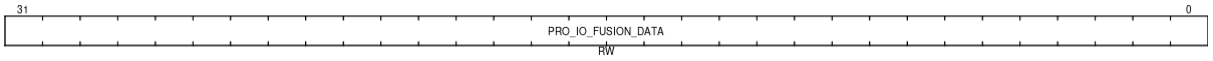


Fig. 6.4.17: PRO_IO_FUSION_DATA

6.4.4 Workflow

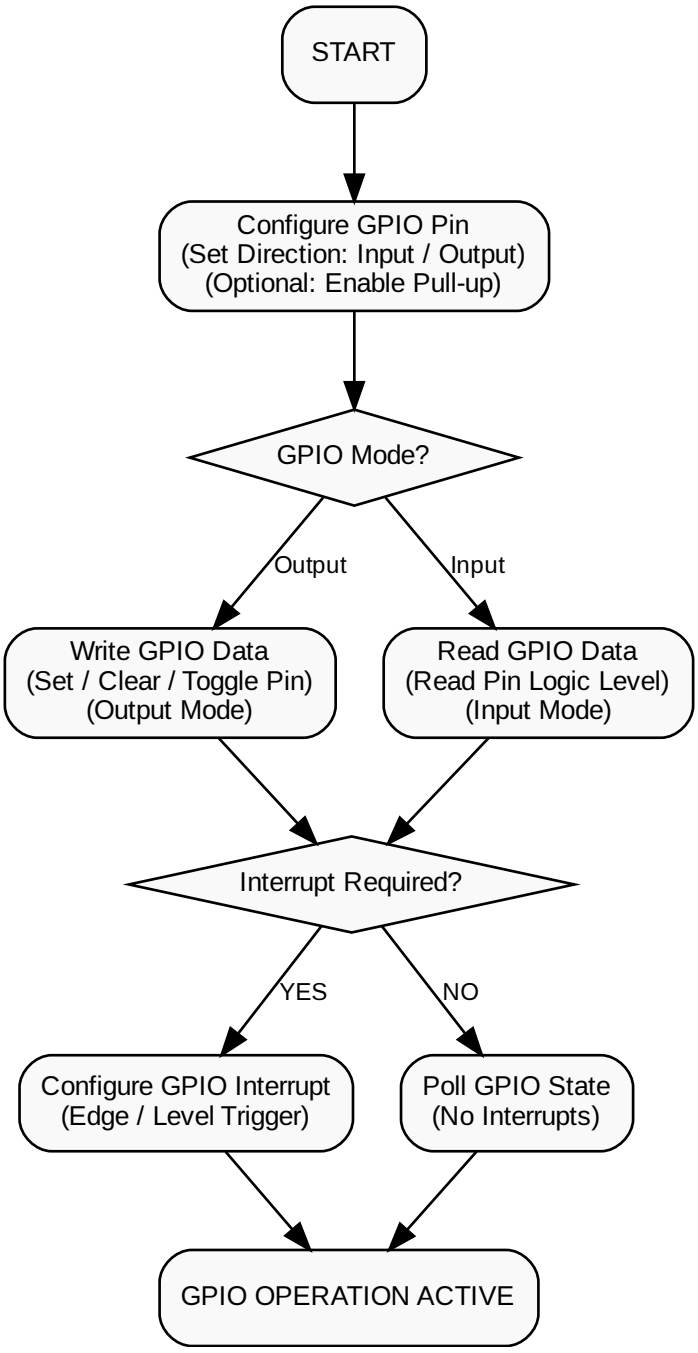
GPIO Operation

1. **Configure GPIO Direction:** Configure each GPIO pin as either an input or an output based on the application. Input pins are used to read external signals such as buttons or sensors, while output pins are used to drive external devices such as LEDs, motors, or buzzers. Internal pull-up resistors may be enabled for input pins connected to mechanical switches to prevent floating states.

2. **Write GPIO Output:** For GPIO pins configured as outputs, write operations control the pin voltage level. Setting the pin HIGH drives the output voltage, while setting it LOW removes the voltage. The pin state can also be toggled to alternate between HIGH and LOW.

3. **Read GPIO Input:** For GPIO pins configured as inputs, read operations return the current logic level of the pin. A value of 0 indicates a LOW level, and a value of 1 indicates a HIGH level, reflecting the state of the connected external signal.

4. **Configure GPIO Interrupts:** GPIO pins may be configured to generate interrupts on specific pin. This allows the processor to respond immediately to external events, such as a button press, without continuous polling.


START
Configure GPIO Pin
(Set Direction: Input / Output)
(Optional: Enable Pull-up)
GPIO Mode?
Output
Input
Write GPIO Data
(Set / Clear / Toggle Pin)
(Output Mode)
Read GPIO Data
(Read Pin Logic Level)
(Input Mode)
Interrupt Required?
YES
NO
Configure GPIO Interrupt
(Edge / Level Trigger)
Poll GPIO State
(No Interrupts)
GPIO OPERATION ACTIVE

ProIO Operation

1. **Select ProIO Type and Direction:** Choose the required data group width (2-bit, 4-bit, 8-bit, or 12-bit) and configure the data direction. In input mode, data is captured from GPIO pins and enqueued into the FIFO. In output mode, data is dequeued from the FIFO and driven onto the GPIO pins.

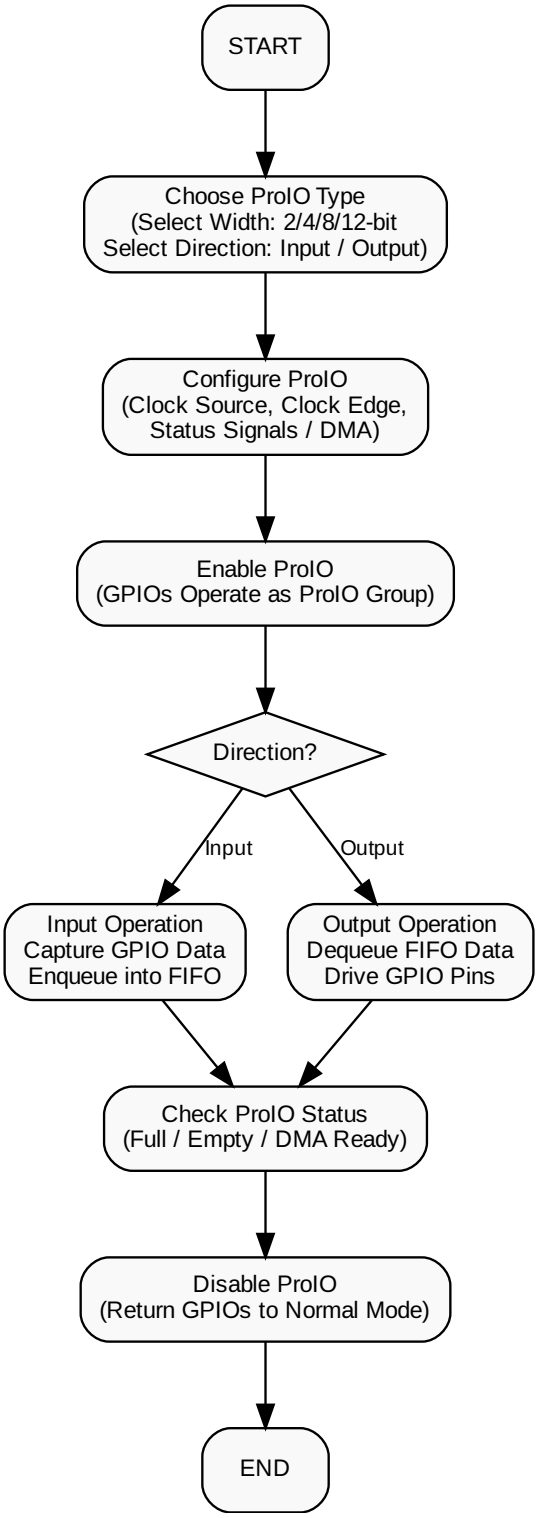
2. **Configure Clock and Transfer Settings:** Select the clock source for the ProIO group, either an internal clock or an external clock provided through a GPIO pin. Configure the active clock edge to define when data is sampled or driven. Status signaling may also be enabled for DMA integration.

3. **Enable ProIO Operation:** Enable the ProIO group to begin operation. Once enabled, the associated GPIO pins operate as a unified data interface rather than individual GPIOs.

4. **Transfer Data:** In input mode, data is captured on each clock edge and stored in the FIFO. In output mode, data is read from the FIFO and driven onto the GPIO pins in synchronization with the clock, supporting continuous high-speed transfers.

5. **ProIO Status:** Check status indicators to determine FIFO availability or readiness. These signals can also be used to trigger DMA transfers without CPU intervention.

6. **Disable ProIO:** Disable the ProIO group after completing data transfers. This stops the ProIO operation and restores the GPIO pins to standard GPIO functionality.


START
Choose ProIO Type
(Select Width: 2/4/8/12-bit
Select Direction: Input / Output)
Configure ProIO
(Clock Source, Clock Edge,
Status Signals / DMA)
Enable ProIO
(GPIOs Operate as ProIO Group)
Direction?
Input
Output
Input Operation
Capture GPIO Data
Enqueue into FIFO
Output Operation
Dequeue FIFO Data
Drive GPIO Pins
Check ProIO Status
(Full / Empty / DMA Ready)
Disable ProIO
(Return GPIOs to Normal Mode)
END

6.5 General Purpose Timer (GPTimer)

General Purpose Timer (GPTimer) is a hardware timer module. It is used to generate accurate time delays and PWM signals. This section describes how to configure and use GPTimer.

Note

GPIO38 to GPIO41 can be configured as GPIO or GPTimer. For more details refer:*Pinmux Register Map*.

6.5.1 Instance Details

This section lists the available peripheral instances along with their base addresses and interrupt IDs.

Table 6.5.1: GPTimer Port Register Map

GPTimer Port	Base Address	Interrupt ID
GPTimer Port0	0x00044200	47
GPTimer Port1	0x00044220	48
GPTimer Port2	0x00044240	49
GPTimer Port3	0x00044260	50

6.5.2 Register Map and Details

This section provides a quick reference to the registers defined in `secure_iot.h`. Each register name links to its detailed description.

Table 6.5.2: GPTimer Register Map

Register Name	Offset (Hex)	Length (Bits)	Description
CTRL	0x00	16	Manages the timer's operational state, mode selection, and status flags. This register is used to start, stop, and monitor the timer.
CLOCK_CTRL	0x04	32	Configures the clock source and prescaler settings, allowing the timer to count at a slower rate by dividing the input clock frequency.
COUNT	0x08	32	Holds the current count value, which increments or decrements based on the timer mode (up, down, or up/down).
RPTD_COUNT	0x0c	32	Specifies the number of times the timer should repeat the count sequence before stopping, used in applications requiring a set number of pulses or intervals.
DUTY_CYCLE	0x10	32	Sets the duty cycle for PWM mode, defining the on/off duration of the output waveform as a percentage of the full period.
PERIOD	0x14	32	Determines the duration of one complete cycle in periodic or PWM mode, controlling the overall timing interval.
CAPTURE_INP	0x18	32	Records the timer count at the moment an external input event occurs, useful for measuring pulse width, frequency, or timing between external events.

Control Register(CTRL Register)

This 16-bit register is used to manage the timer's operational state, mode selection, and status flags.

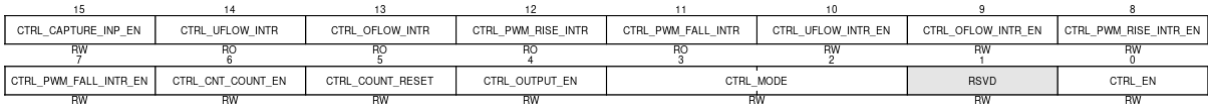


Fig. 6.5.1: GPTimer Control Register

Table 6.5.3: GPT register field description

Bits	Field Name	Attribute	Description
[0:0]	CTRL_EN	RW	If set, enables the gptimer.
[1:1]	RSVD	RW	Reserved for future use.
[2:3]	CTRL_MODE	RW	**Timer mode selection.** • **00**: PWM operation • **01**: Up counter • **10**: Down Counter • **11**: UpDown Counter
[4:4]	CTRL_OUTPUT_EN	RW	If set, enables the output wave.
[5:5]	CTRL_COUNT_RESET	RW	If set, resets the counter, registers and interrupts.
[6:6]	CTRL_CNT_COUNT_EN	RW	If set, enables continuous counting.
[7:7]	CTRL_PWM_FALL_INTR_E	RW	If set, enables PWM fall interrupt.
[8:8]	CTRL_PWM_RISE_INTR_EI	RW	If set, enables PWM rise interrupt.
[9:9]	CTRL_OFLOW_INTR_EN	RW	If set, enables counter overflow interrupt.
[10:10]	CTRL_UFLOW_INTR_EN	RW	If set, enables counter underflow interrupt.
[11:11]	CTRL_PWM_FALL_INTR	RO	This bit shows the status of the PWM rise interrupt.
[12:12]	CTRL_PWM_RISE_INTR	RO	This bit shows the status of the PWM fall interrupt.

continues on next page

Table 6.5.3 – continued from previous page

Bits	Field Name	Attribute	Description
[13:13]	CTRL_OFLOW_INTR	RO	Indicates when the up-counter exceeds the period value.
[14:14]	CTRL_UFLOW_INTR	RO	Indicates when the down-counter goes below zero.
[15:15]	CTRL_CAPTURE_INP_EN	RW	If set, captures counter value when this bit matches the input signal(0 or 1).

Clock Control Register(CLOCK_CTRL)

This 32-bit Register is responsible for configuring the clock settings for GPTimer, including the input clock source and prescaler values. By adjusting these parameters, the register allows precise control over the timer's counting rate, enabling flexible timing configurations across different operating modes.

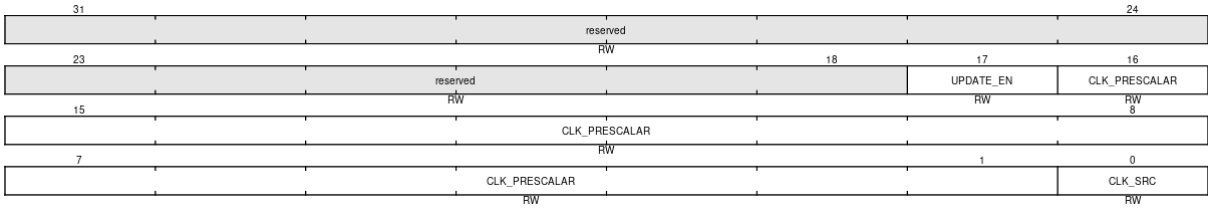


Fig. 6.5.2: GPTimer Clock Control Register

Table 6.5.4: Clock Control Register Fields

Bits	Field Name	Attribute	Description
[0:0]	CLK_SRC	RO	Clock source selection(Read only(0))
[1:16]	CLK_PRESCALAR	RW	Prescaler for clock division.

continues on next page

Table 6.5.4 – continued from previous page

Bits	Field Name	Attribute	Description
[17:17]	UPDATE_EN	RW	This bit has to be cleared by software in order to update period and duty cycle values.
[18:31]	RESERVED	RW	Reserved for future use.

Counter Register(COUNT)

This 32-bit register stores the current count value of the timer. Depending on the mode selected for the timer, the counter behaves differently.

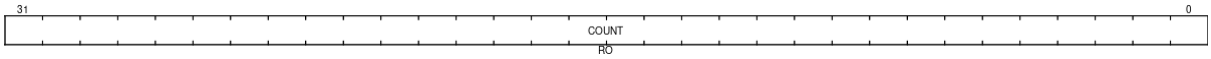


Fig. 6.5.3: GPTimer Counter Register

It is automatically updated by the timer hardware as it counts according to the selected mode, and the count is compared with the *period* value for various events (such as overflow or underflow).

Table 6.5.5: Counter Register Modes

Mode	Description
PWM and Up Counter	Counter starts at 0 and counts up until it reaches the *period* value.
Down Counter	Counter starts at *period-1* and counts down to 0.
UpDown Counter	Counter starts at 0 and counts up until it reaches the *period* value again counts down from *period-1* to 0.

Repeated Count Register(RPTD_COUNT)

This 32-bit register tracks the number of times the counter has repeated its count cycle. It is initialized to zero and increments each time the counter restarts, provided

the *continous_count* variable is set to high. At the end of the counting process, the *Repeated_count* register holds the total count of repetitions.

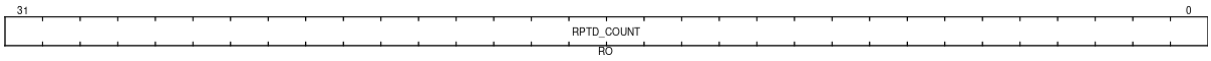


Fig. 6.5.4: `GPTimer` Repeated Count Register

DUTY_CYCLE Register

The Duty Cycle register holds a 32-bit duty cycle value specifically for PWM mode. This register controls the duration for which the output remains high, depending on the value stored in Duty cycle register.

Fig. 6.5.5: `GPTimer` Duty Cycle Register

PERIOD Register

The *period* register stores a 32-bit period value that defines the period for the counter. The counter increments or decrements based on the value stored in the *period* register.

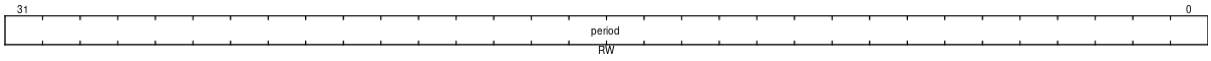


Fig. 6.5.6: GPTimer Period Register

Capture Input Register(CAPTURE_INP)

The *capture_input* bit is located in the CTRL Register. The *in/out* bit, which has a two-way function, provides the input (0 or 1) and receives the PWM output. When the input from the *in/out* bit matches the *capture_input* bit (either 0 or 1), the current value of the *counter* is copied into the *Capture Input* register.

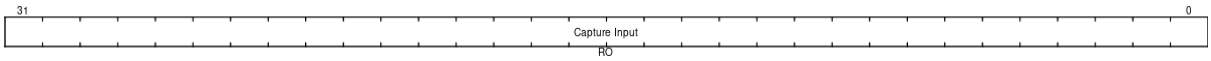
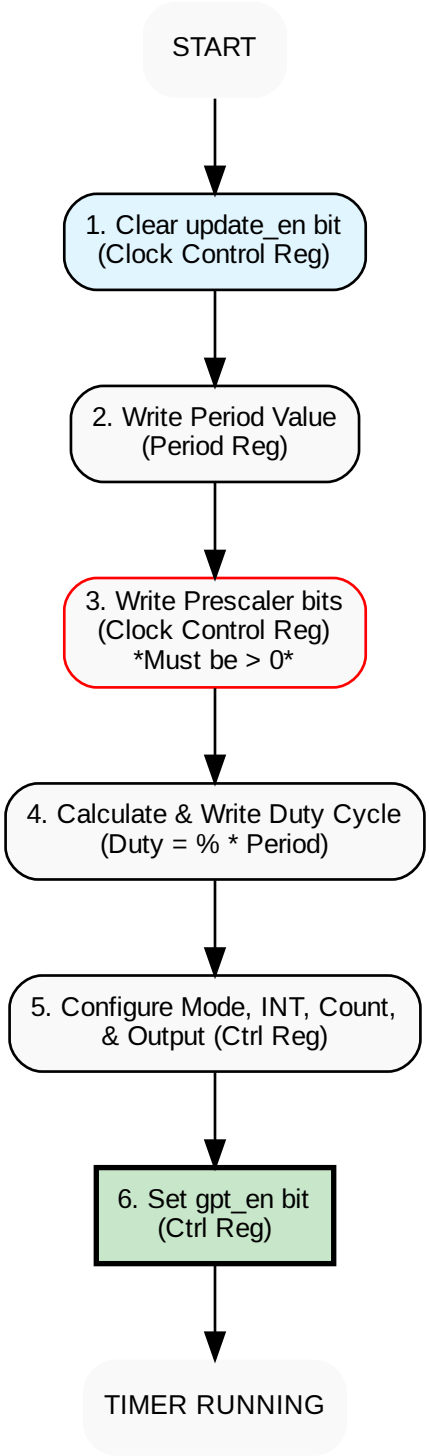


Fig. 6.5.7: GPTimer CAPTURE_INP Register

6.5.3 Workflow

1. **Clear Update_en bit** : The update_en bit in the clock control register has to be cleared by software in order to update period and duty cycle value to the respective period and duty cycle register.
2. **Write Period value** : The next step is to update the period value to period register.
3. **Check Prescaler value** : If prescaler value is zero then invalid prescaler error, else the prescaler value has to be written to the clock control register.
4. **Write Duty Cycle value** : The duty cycle value should be written to the duty cycle register. This register accepts values only from 0 to 100 and make sure to multiply the duty cycle value with period before writing it to the register.
5. **Write Configuration parameters** : Write the specific configuration settings such as mode selection, interrupt enable, continuous count enable, output enable to the control register.
6. **Set the gpt_en bit** : To enable gptimer, set the gpt_en bit in control register.


START
1. Clear update_en bit
(Clock Control Reg)
2. Write Period Value
(Period Reg)
3. Write Prescaler bits
(Clock Control Reg)
Must be > 0
4. Calculate & Write Duty Cycle
(Duty = % * Period)
5. Configure Mode, INT, Count,
& Output (Ctrl Reg)
6. Set gpt_en bit
(Ctrl Reg)
TIMER RUNNING

6.6 Inter-Integrated Circuit (I2C)

I2C is a serial, synchronous, half-duplex communication protocol that supports multiple devices on the same bus, allowing multiple masters and slaves to coexist. It operates using two bidirectional, open-drain lines: the serial data line (SDA) and the serial clock line (SCL), both of which are pulled high by resistors. S2401 has two I2C controllers, each responsible for managing communication on the I2C bus. Each I2C controller can be configured only as master. I2C slave devices typically have a 7-bit or 10-bit address. S2401 supports I2C, with data transfer speed up to 1 MHz.

6.6.1 I2C Instance Details

This section lists the available I2C instances along with their base addresses and interrupt IDs.

Table 6.6.1: I2C Port Register Map

I2C Port	Base Address	Interrupt ID
I2C0	0x00044000	51
I2C1	0x00044100	52

Note

The interrupt IDs listed above are defined but not used.

6.6.2 Register Map and Details

This section provides a quick reference to the register map with the offsets for I2C instance. Each register name links to its detailed description.

Table 6.6.2: I2C Register Map

Register Name	Offset	Description
S2	0x00	Used to set I2C Prescaler value.
CTRL	0x08	Used to control state of transaction.
S0	0x10	Used to set slave address, send and receive data.
STATUS	0x18	Used to view the status of I2C transaction.
SCL	0x38	Used to set I2C Clock division value.

S2 (Prescaler) Register

The **S2** is an 8-bit register, that holds the prescaler value.

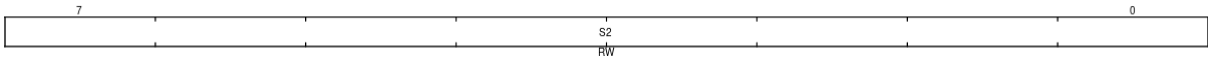


Fig. 6.6.1: *I2C S2*

CTRL (Control) Register

The **CTRL** is an 8-bit register, that is used to manage several I2C states.

Fig. 6.6.2: *I2C CTRL*

Table 6.6.3: CTRL Register Fields

Bits	Field Name	Permission	Description
[0:0]	CTRL_ACK	WO	Used to set ACK in slave mode of I2C for future use.

continues on next page

Table 6.6.3 – continued from previous page

Bits	Field Name	Permission	Description
[1:1]	CTRL_STO	WO	When this bit is set, a stop bit is transmitted.
[2:2]	CTRL_STA	WO	When this bit is set, a start bit is transmitted.
[3:3]	CTRL_ENI	WO	Enables the external interrupt output.
[5:4]	RESERVED	RW	Reserved for future use.
[6:6]	CTRL_ESO	WO	When this bit is set, enables the serial interface.
[7:7]	CTRL_PIN	WO	When this bit is set, all status bits are reset to 0.

S0 (Data) Register

The **S0** is an 8-bit register, that holds the I2C data to be received or to be transmitted.

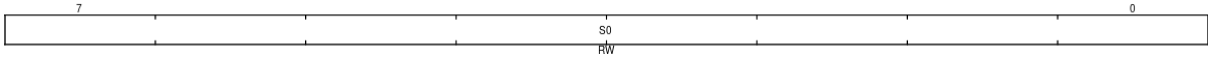


Fig. 6.6.3: *I2C S0*

STATUS Register

The **STATUS** is an 8-bit register, that holds the flags indicating the current state of I2C peripheral transmission, reception and bus.

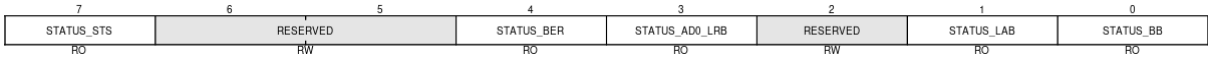


Fig. 6.6.4: *I2C STATUS*

Table 6.6.4: STATUS Register Fields

Bits	Field Name	Permission	Description
[0:0]	STATUS_BB	RO	The busy bit indicates the bus status, where 0 means the bus is busy and 1 means the bus is free.
[1:1]	STATUS_LAB	RO	If bit is high, indicates the arbitration is lost to another master on the I2C bus in multi-master mode.
[2:2]	RESERVED	RW	Reserved for future use.
[3:3]	STATUS_AD0_LRI	RO	Indicates the last bit received on the I2C bus. It is used to check the ACK received from the slave.
[4:4]	STATUS_BER	RO	Indicates bus error, a misplaced START or STOP condition has been detected.
[6:5]	RESERVED	RW	Reserved for future use.
[7:7]	STATUS_STS	RO	The pin bit that is polled to determine whether the transmission/reception of a byte is complete.

SCL (Clock division) Register

The **SCL** is an 32-bit register, that is used to set the Clock division value for I2C peripheral.

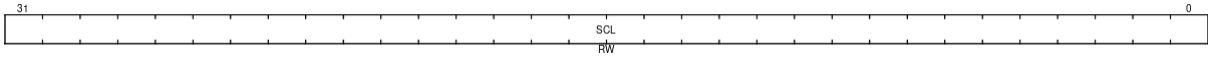


Fig. 6.6.5: *I2C SCL*

SCL Frequency Calculation :

The SCL is generated by dividing the system clock (*CLOCK_FREQUENCY*) using the prescaler (*S2*) and I2C clock frequency (*I2C_CLOCK_FREQUENCY*) parameters.

$$SCL = \frac{CLOCK\_FREQUENCY}{2 * (S2 + 1)(I2C\_CLOCK\_FREQUENCY)}$$

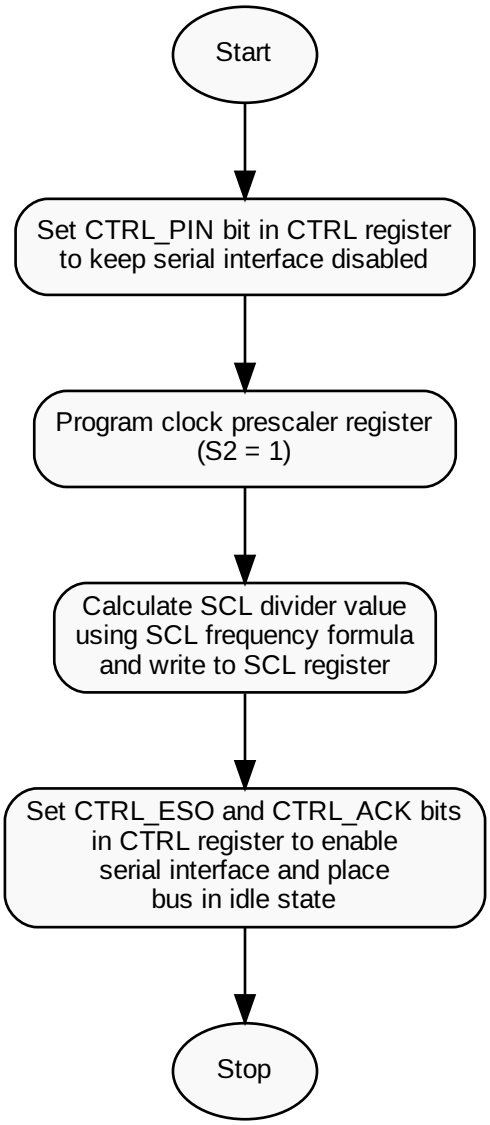
$$0 \leq S2 \leq 2^8 - 1$$

6.6.3 Workflow

I2C initialization Flow

1. Set the *CTRL_PIN* bit in the control register (*CTRL*) to keep the serial interface disabled during configuration.
2. Program the clock prescaler register (*S2*) with the value 1 to set the timing base.
3. Calculate the SCL divider value using the above-mentioned *SCL Frequency Calculation* and write it to the *SCL* register.
4. Set the *CTRL_ESO* and *CTRL_ACK* bits in the control register (*CTRL*) to enable the serial interface and place the bus in the idle state.
5. I^2C initialization is complete, and the bus is ready for operation.

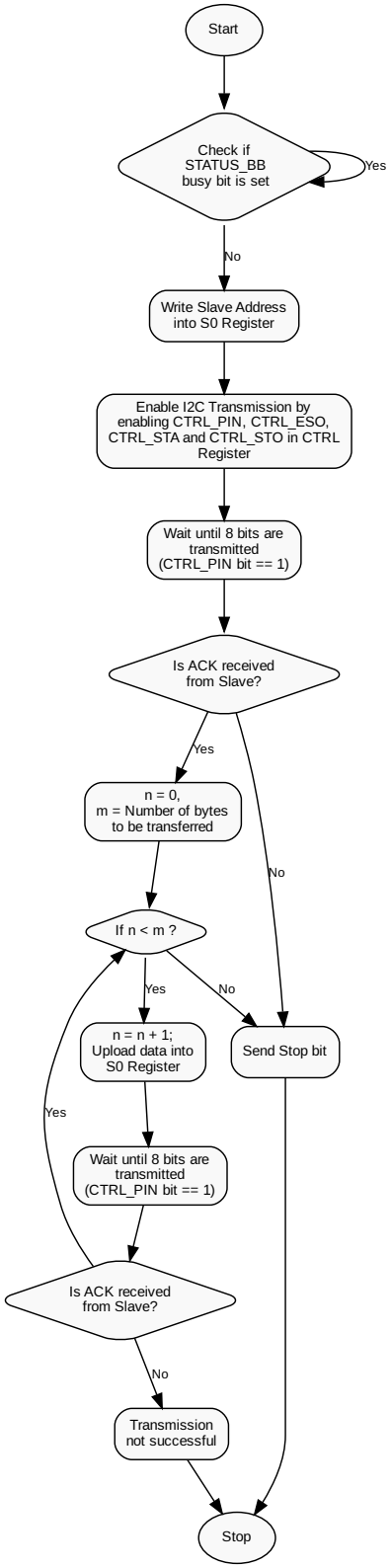
The following flowchart describes the basic Initialization process.



I2C Transmit Flow

1. Start the I^2C master transmit operation.
2. Check the bus busy status bit (*STATUS_BB*); if the bus is free, proceed with transmission, otherwise wait until the bus becomes free.
3. Write the slave address into the data register (*S0*).

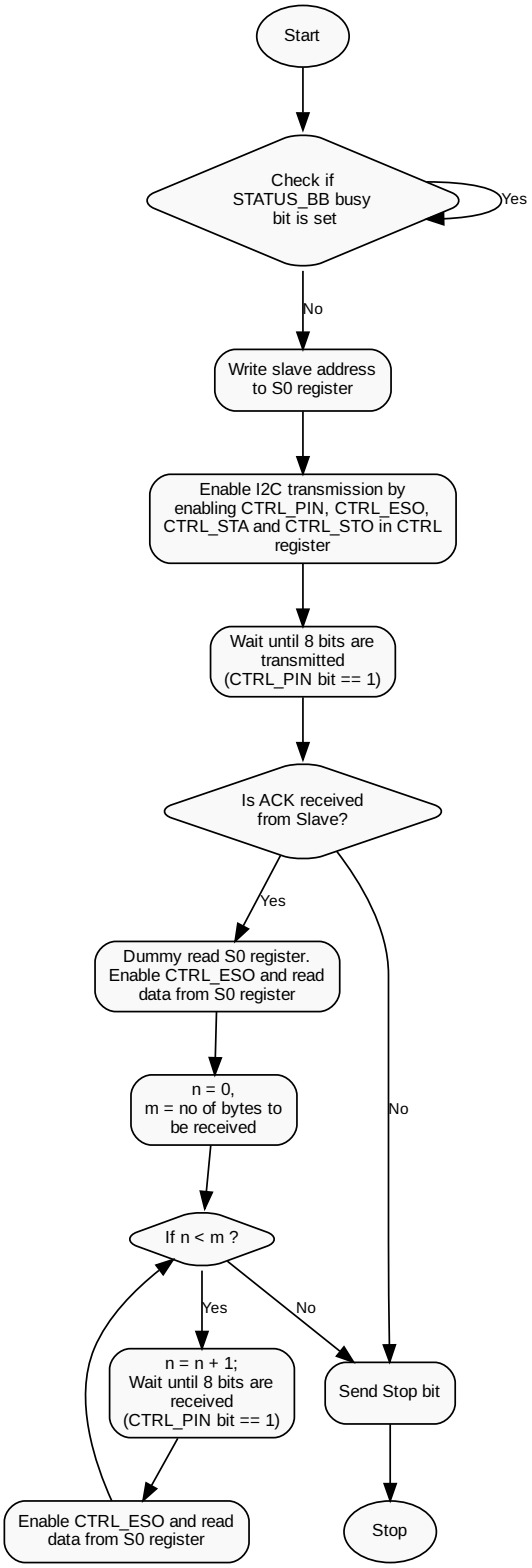
4. To start the I^2C transmission, enable START by setting *CTRL_PIN*, *CTRL_ESO*, *CTRL_STA* and *CTRL_ACK* bits in the control register (*CTRL*).
5. Wait until 8 bits are transmitted, as indicated by the PIN bit (*CTRL_PIN*) .
6. Check for slave ACK via *STATUS_AD0_LRB*; if high, issue a STOP condition by setting *CTRL_PIN*, *CTRL_ESO*, *CTRL_STO*, and *CTRL_ACK* bits and terminate.
7. Write the data byte into the data register (*S0*).
8. Wait until 8 bits are transmitted, PIN bit is high.
9. Check for slave ACK after data transmission, if high continue transmitting remaining data bytes else issue a STOP condition and terminate.
10. After all data bytes are transmitted, based on the mode generate a STOP or REP-START condition.
11. End the I^2C master transmit operation.


Start
Check if STATUS_BB busy bit is set
Yes
No
Write Slave Address into S0 Register
Enable I2C Transmission by enabling CTRL_PIN, CTRL_ESO, CTRL_STA and CTRL_STO in CTRL Register
Wait until 8 bits are transmitted (CTRL_PIN bit == 1)
Is ACK received from Slave?
Yes
No
n = 0, m = Number of bytes to be transferred
If n < m ?
Yes
No
n = n + 1; Upload data into S0 Register
Send Stop bit
Yes
Wait until 8 bits are transmitted (CTRL_PIN bit == 1)
Is ACK received from Slave?
No
Transmission not successful
Stop

I2C receive Flow

1. Start the I^2C master receive operation.
2. Check the bus busy status bit (*STATUS_BB*); if the bus is free, proceed with the operation, otherwise wait until the bus becomes free.
3. Write the 7-bit slave address with the read bit set into the data register (*S0*).
4. Enable START by setting *CTRL_PIN*, *CTRL_ESO*, *CTRL_STA*, and *CTRL_ACK* bits in the control register (*CTRL*).
5. Wait until 8 bits of the slave address are transmitted, as indicated by the *CTRL_PIN* bit.
6. Check for slave ACK via *STATUS_AD0_LRB*; if not received, issue a STOP condition by setting *CTRL_PIN*, *CTRL_ESO*, *CTRL_STO*, and *CTRL_ACK* bits and terminate.
7. Perform a dummy read of the *S0* register to initialize the receive process.
8. Wait until 8 bits are received by polling (*CTRL_PIN* == 1), continue for all remaining bytes.
9. After receiving all bytes, send NACK to slave by enabling *CTRL_ESO* bit.
10. Based on the mode, generate a STOP condition or a repeated START condition.
11. End the I^2C master receive operation.

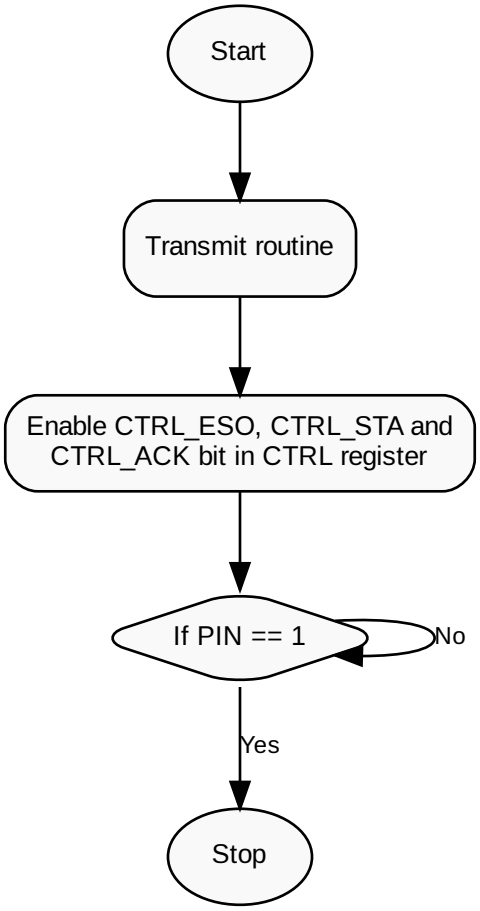
The following flowchart describes the basic Receive process.


Start
Check if STATUS_BB busy bit is set
Yes
No
Write slave address to S0 register
Enable I2C transmission by enabling CTRL_PIN, CTRL_ESO, CTRL_STA and CTRL_STO in CTRL register
Wait until 8 bits are transmitted (CTRL_PIN bit == 1)
Is ACK received from Slave?
Yes
No
Dummy read S0 register. Enable CTRL_ESO and read data from S0 register
n = 0, m = no of bytes to be received
If n < m ?
Yes
No
n = n + 1; Wait until 8 bits are received (CTRL_PIN bit == 1)
Enable CTRL_ESO and read data from S0 register
Send Stop bit
Stop

I2C Repeatedstart Flow

1. Enable the *CTRL_ESO*, *CTRL_STA*, and *CTRL_ACK* bits in the control register (*CTRL*) to generate a repeated start condition.
2. Check the *CTRL_PIN* status; if set, it indicates that the repeated start condition has been completed; otherwise, continue waiting.
3. After the repeated start, perform the necessary I^2C transmit or receive operation.
4. Once the operation is completed, issue a STOP condition to end the repeated start operation.

The following flowchart describes the basic Repeated start process.



6.7 Pin Multiplexing (Pinmux)

Pinmux (Pin Multiplexing) is a mechanism that allows a single physical pin to support multiple alternate functions. To use more internal peripheral signals than available pins, each pin can be configured to operate as different interfaces such as GPIO, UART, SPI, PWM, JTAG, etc. Through pinmux configuration, one can select which function is active by assigning the required alternate function to the pin. Only one function can be enabled on a pin at a time, to avoid conflicts between peripherals. Pinmux increases design flexibility and optimizes the use of limited pin resources in embedded systems.

6.7.1 Instance Details

There is only one instance which is used to configure the Pinmux.

Table 6.7.1: Pinmux Instance Map

Pinmux Instance	Base Address (Hex)
Pinmux	0x00040400

6.7.2 Register Map and details

The table below lists the registers of the PINMUX, along with their addresses and descriptions.Each register is 32-bit wide.

Table 6.7.2: Pinmux Register Map

Register Name	Offset (Hex)	Function when clear	Fucntion when set
MUX0	0x0000	GPIO0	PWM0
MUX1	0x0004	GPIO1	PWM1
MUX2	0x0008	GPIO2	PWM2

continues on next page

Table 6.7.2 – continued from previous page

Register Name	Offset (Hex)	Function when clear	Fucntion when set
MUX3	0x000C	GPIO3	PWM3
MUX4	0x0010	GPIO4	PWM4
MUX5	0x0014	GPIO5	PWM5
MUX6	0x0018	GPIO6	PWM6
MUX7	0x001C	GPIO7	PWM7
MUX8	0x0020	GPIO17	PWM8
MUX9	0x0024	GPIO18	PWM9
MUX10	0x0028	GPIO19	PWM10
MUX11	0x002C	GPIO20	PWM11
MUX12	0x0030	GPIO21	PWM12
MUX13	0x0034	GPIO22	PWM13
MUX14	0x0038	SPI2_MOSI	GPIO32
MUX15	0x003C	SPI2_MISO	GPIO33
MUX16	0x0040	SPI2_NCS	GPIO34
MUX17	0x0044	SPI3_MOSI	GPIO35
MUX18	0x0048	SPI3_MISO	GPIO36
MUX19	0x004C	SPI3_NCS	GPIO37
MUX20	0x0050	GPIO8	UART3_TX
MUX21	0x0054	GPIO9	UART3_RX
MUX22	0x0058	GPIO11	UART4_TX
MUX23	0x005C	GPIO15	UART4_RX
MUX24	0x0060	GPTIMER0	GPIO38
MUX25	0x0064	GPTIMER1	GPIO39
MUX26	0x0068	GPTIMER2	GPIO40

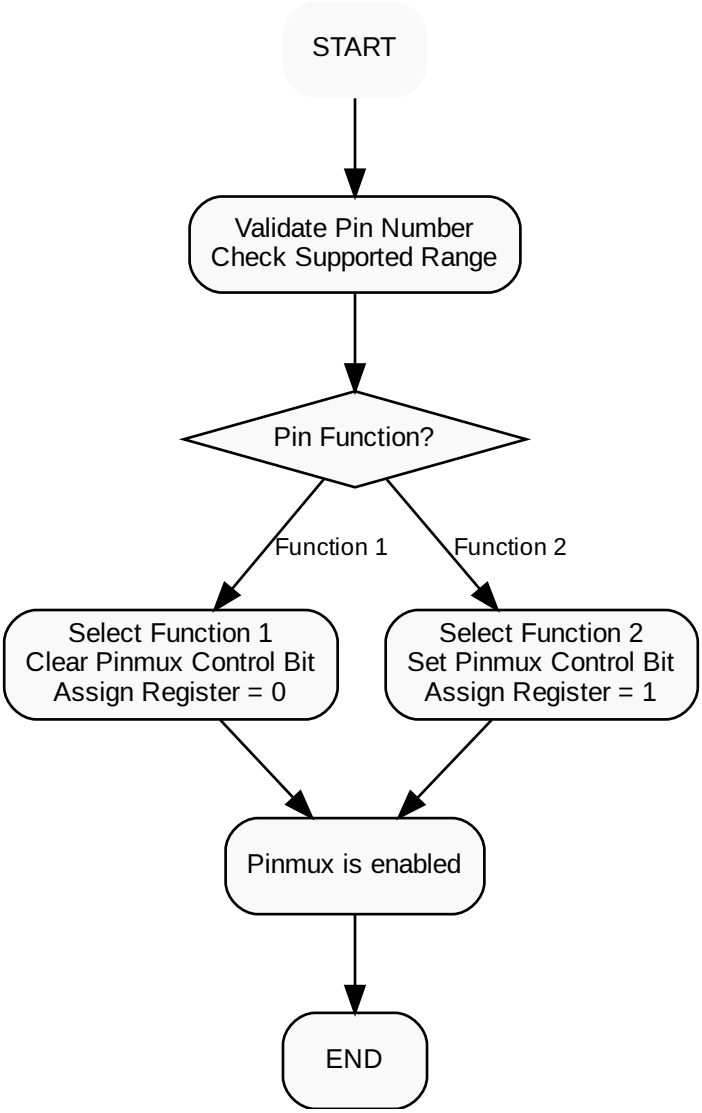
continues on next page

Table 6.7.2 – continued from previous page

Register Name	Offset (Hex)	Function when clear	Fucntion when set
MUX27	0x006C	GPTIMER3	GPIO41
MUX28	0x0070	JTAG_TDI	GPIO42
MUX29	0x0074	JTAG_TMS	GPIO43
MUX30	0x0078	JTAG_TDO	GPIO44

6.7.3 Workflow

1. Validate the pin number to ensure it is within the supported pin range.
2. Select the required pin function for the pin (Function 1 or Function 2).
3. If Function 1 is selected, clear the corresponding pinmux register.
4. If Function 2 is selected, set the corresponding pinmux register.



6.8 Platform Level Interrupt Controller (PLIC)

The Platform Level Interrupt Controller (PLIC) manages external interrupt requests from on-chip peripherals and delivers them to the core. It provides centralized interrupt control through configurable prioritization, masking, and software-controlled interrupt handling. The PLIC supports multiple interrupt sources and routes active interrupts to

the core based on their configured priority.

Key Features:

- Support for up to 81 interrupt sources.
- Configurable per-interrupt priority levels.
- Per-interrupt enable and pending status control.

6.8.1 Instance Details

The system contains a single PLIC instance with a fixed base address.

Table 6.8.1: `PLIC` Instance Map

Instance	Base Address	Interrupt ID
PLIC	0x0C000000	NA

6.8.2 Interrupt Identifiers (IDs)

Each interrupt source is assigned an unsigned integer identifier, beginning at the value 1. The PLIC supports interrupt IDs in the range 1-81, where each ID corresponds to a unique interrupt source. An interrupt ID of 0 is reserved to mean “no interrupt”.

6.8.3 Register Map and Details

This section provides a quick reference to the registers defined in `secure_iot.h`. Each register name links to its detailed description.

Table 6.8.2: PLIC Register Map

Register Name	Offset	Length (Bits)	Description
PRIORITY (Interrupt Priority Register)	0x0000	32	Defines the priority level for each interrupt source. One priority register is provided per interrupt ID.
PENDING_INTR (Interrupt Pending Register)	0x1000	32	Indicates the pending status of interrupt sources. Each bit corresponds to an interrupt ID.
INTR_ENABLE (Interrupt Enable Register)	0x2000	64	Controls the enable status of interrupt sources. Each bit enables or disables the corresponding interrupt ID.
PRIORITY_THRES (Priority Threshold Register)	0x200000	32	Specifies the minimum priority required for an interrupt to be forwarded to the core.
INTR_COMPLET (Interrupt Claim/Completi Register)	0x200004	32	Used to claim the highest-priority pending interrupt and to signal completion of interrupt servicing.

Interrupt Priority Register (PRIORITY)

Each interrupt source can be assigned a priority by writing to its 32-bit memory-mapped priority register (ie. PRIORITY[Interrupt_ID]) .

- The PLIC supports seven priority levels.
- Priority values range from **0** to **6**, where a higher value represents a higher interrupt priority.
- A priority value of **0** is reserved to mean “never interrupt” and effectively disables the interrupt.
- When multiple interrupts have the same priority, the interrupt with the lowest in-

terrupt ID is serviced first.

Table 6.8.3: Interrupt Priority Registers

Register Name	Offset	Description
PRIORITY[0]	0x0000 (reserved)	Interrupt Priority Register for interrupt ID 0.
PRIORITY[1]	0x0004	Interrupt Priority Register for interrupt ID 1.
PRIORITY[2]	0x0008	Interrupt Priority Register for interrupt ID 2.
...	...	...
PRIORITY[81]	0x0144	Interrupt Priority Register for interrupt ID 81.

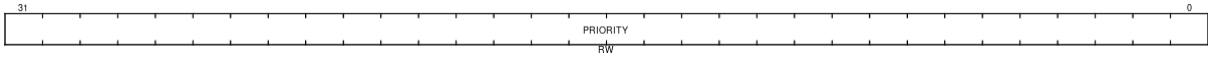


Fig. 6.8.1: PLIC Interrupt Priority Register

Interrupt Pending Register (PENDING_INTR)

Each interrupt source has a dedicated pending bit in the PLIC. When an interrupt source generates an interrupt, the PLIC sets the corresponding pending bit to '1', indicating that the interrupt is active and waiting to be handled. The pending bit remains set until the interrupt is acknowledged and serviced, after which the PLIC clears it. All interrupt pending bits in this register are read-only (RO).

- The interrupt pending registers are arranged as a contiguous array of 4-bytes (32-bits) registers.
- The interrupt pending bit for interrupt ID *N* is stored in bit position *N mod 32* within the corresponding 32-bit register.
- Bit *0* of the first 4 bytes represents the *non-existent interrupt ID 0*, which is hard-wired to 0.

Table 6.8.4: Interrupt Pending Registers

Register Name	Offset	Description
PENDING_0_31	0x1000	Interrupt pending bit for sources 0 to 31.
PENDING_32_63	0x1004	Interrupt pending bit for sources 32 to 63.
PENDING_64_81	0x1008	Interrupt pending bit for sources 64 to 81.

Note

The `PENDING_64_81` register implements pending bits only for interrupt IDs **64-81** (18 bits). All other bits in this register are reserved.

Interrupt Enable Register (INTR_ENABLE)

Each interrupt source can be enabled by setting its corresponding bit in the Interrupt Enable Register. An interrupt is disabled by clearing its corresponding bit in the Interrupt Enable register. All interrupt enable bits in this register are read/write (RW).

- The interrupt enable registers are arranged as a contiguous array of 8-bytes (64-bits) registers.
- The enable bit for interrupt ID *N* is stored in bit position *N mod 64* within the corresponding 64-bit register.
- Bit *0* of the first 8 bytes represents the *non-existent interrupt ID 0*, which is hard-wired to 0.

Table 6.8.5: Interrupt Enable Register

Register Name	Offset	Description
INTR_EN_0_63	0x2000	Interrupt Enable bit for sources 0 to 63.
INTR_EN_64_81	0x2008	Interrupt Enable bit for sources 64 to 81.

Note

The `INTR_EN_64_81` register implements enable bits only for interrupt IDs **64-81** (18 bits). All other bits in this register are reserved.

Priority Threshold Register (PRIORITY_THRES)

The Priority Threshold Register is used to set the threshold priority level. The threshold register supports 7 priority levels. Interrupts with a priority lesser than or equal to the threshold are masked and will not be forwarded. For example, a threshold value of zero, permits all interrupts with non-zero priority.

Fig. 6.8.2: PLIC Interrupt Threshold Register

Interrupt Claim/Completion Register (INTR_COMPLETE)

The Interrupt Claim/Complete register supports two operations depending on access type:

- **Read (Claim)**: Claims the highest-priority pending interrupt.
- **Write (Complete)**: Signals completion of a previously claimed interrupt.

Claim Operation (Read)

Reading this register returns the interrupt ID of the highest-priority pending interrupt.

- If no interrupt is pending, the register returns **0**.
- A successful claim **automatically clears** the corresponding pending bit in the Interrupt Pending Register.
- Interrupt claims may be performed at any time, independent of the *MEIP* bit in the *MIP* register is set.

Complete Operation (Write)

Writing an interrupt ID to this register signals completion of interrupt servicing.

- If the completion ID does not match an interrupt source that is currently enabled for the target, the completion is silently ignored.
- Completion allows the PLIC to re-assert the interrupt if it becomes pending again.

Fig. 6.8.3: PLIC Interrupt Claim/Completion Register

6.8.4 Workflow

This section presents the configuration flow for PLIC initialization and interrupt handling.

Configuration Sequence

1. Global Initialization

The PLIC is placed in a known state before configuring individual interrupt sources.

- All interrupt enable registers are cleared.
- A default priority threshold is configured.
- Global machine-level interrupts are enabled at the core.

2. Interrupt Handler Registration

Each interrupt source intended to be used is associated with a software handler.

- A handler function and optional argument are registered for the interrupt ID.
- If no handler is registered, a default handler is invoked.

3. Interrupt Priority Configuration

Each interrupt source intended to be used is assigned an individual priority.

- Higher numeric values represent higher priority.
- Priority configuration must be completed before enabling the interrupt.

4. Interrupt Enable

After the handler and priority are configured, the corresponding interrupt source is enabled.

- Enabled interrupts can become pending when asserted by the peripheral.
- Disabled interrupts are ignored regardless of their priority.

Once enabled, the interrupt source is fully configured and eligible for servicing.

Interrupt Handling

1.Interrupt Claim

When an interrupt is taken, software reads the Interrupt Claim register to obtain the interrupt ID of the highest-priority pending source.

- A value of zero indicates that no interrupt is pending.
- Reading the claim register implicitly acknowledges the interrupt.

2.Interrupt Service Routine Execution

The registered handler corresponding to the claimed interrupt ID is executed.

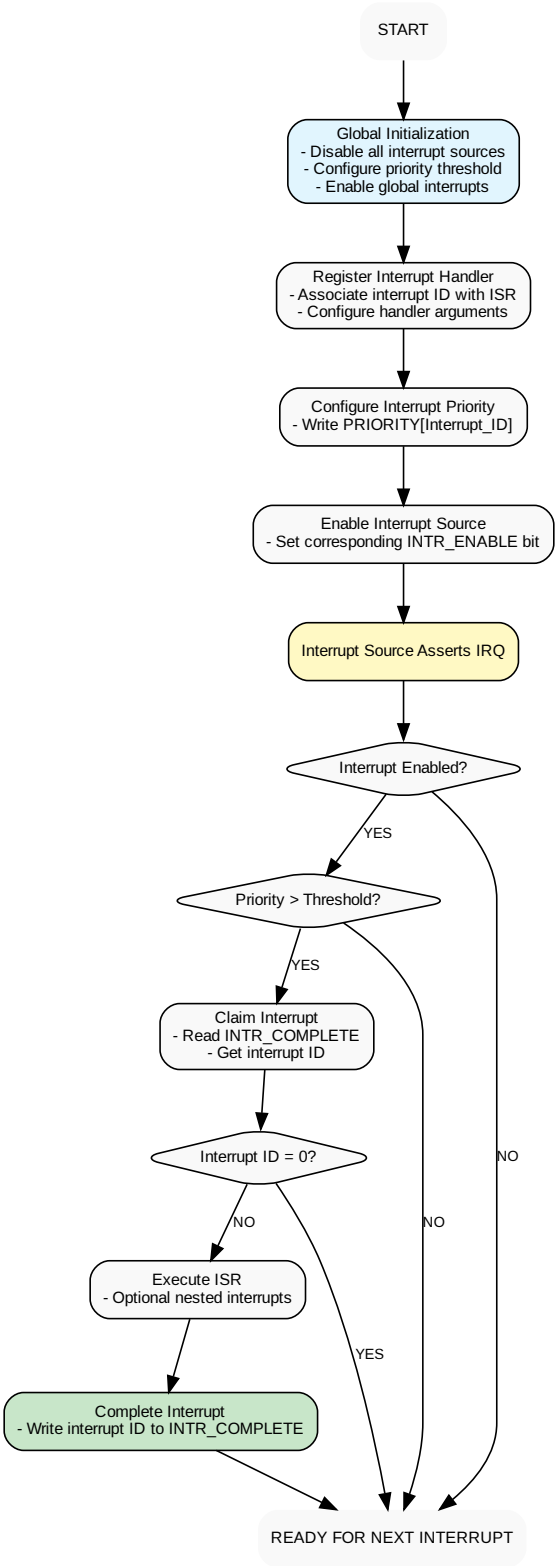
- Optional nested interrupt handling may be enabled, allowing higher-priority interrupts to preempt the current ISR.
- Nested interrupt behavior is controlled by software.

3.Interrupt Completion

After the handler finishes execution, software writes the same interrupt ID to the Interrupt Completion register.

- This signals the PLIC that servicing is complete.
- The interrupt source may be re-asserted if the peripheral condition persists.

Completion is mandatory to allow subsequent interrupts to be serviced.


START
Global Initialization
- Disable all interrupt sources
- Configure priority threshold
- Enable global interrupts
Register Interrupt Handler
- Associate interrupt ID with ISR
- Configure handler arguments
Configure Interrupt Priority
- Write PRIORITY[Interrupt_ID]
Enable Interrupt Source
- Set corresponding INTR_ENABLE bit
Interrupt Source Asserts IRQ
Interrupt Enabled?
YES
NO
Priority > Threshold?
YES
NO
Claim Interrupt
- Read INTR_COMPLETE
- Get interrupt ID
Interrupt ID = 0?
NO
YES
Execute ISR
- Optional nested interrupts
Complete Interrupt
- Write interrupt ID to INTR_COMPLETE
READY FOR NEXT INTERRUPT

6.9 Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is a technique used to control output signals by varying the duty cycle of a digital waveform while keeping the frequency constant. Secure-IoT devices provide 14 configurable PWM channels, PWM0 through PWM13, which can be mapped to GPIO pins using pin multiplexing.

Note

The GPIO0–GPIO7 can be configured as PWM0–PWM7, and GPIO17–GPIO22 can be configured as PWM8–PWM13 using pin-mux respectively.

6.9.1 Instance Details

Table 6.9.1: PWM Instance Map

PWM Instance	Base Address (Hex)	Interrupt line
PWM0	0x00030000	PWM0_IRQn
PWM1	0x00030100	PWM1_IRQn
PWM2	0x00030200	PWM2_IRQn
PWM3	0x00030300	PWM3_IRQn
PWM4	0x00030400	PWM4_IRQn
PWM5	0x00030500	PWM5_IRQn
PWM6	0x00030600	PWM6_IRQn
PWM7	0x00030700	PWM7_IRQn
PWM8	0x00030800	PWM8_IRQn
PWM9	0x00030900	PWM9_IRQn
PWM10	0x00030A00	PWM10_IRQn
PWM11	0x00030B00	PWM11_IRQn
PWM12	0x00030C00	PWM12_IRQn

continues on next page

Table 6.9.1 – continued from previous page

PWM Instance	Base Address (Hex)	Interrupt line
PWM13	0x00030D00	PWM13_IRQn

6.9.2 Register Map and Details

This section consists of the available instances and their corresponding addresses for each PWM.

Table 6.9.2: PWM Register Map

Register Name	Offset (Hex)	Bit	Description
CLOCK_CT	0x0000	16	Configures PWM clock source and frequency.
CTRL	0x0004	16	Enables and sets PWM operation parameters.
PERIOD	0x0008	32	Defines the PWM period.
DUTY_CY(	0x000C	32	Sets the PWM signal's duty cycle.
DEAD-BAND_DEI	0x0010	16	Adds delay to prevent signal overlap.

CLOCK CONTROL Register

This 16-bit register configures the clock source and prescaler for the PWM module.Before enabling the PWM, configure the clock source and prescaler to set the desired PWM frequency.

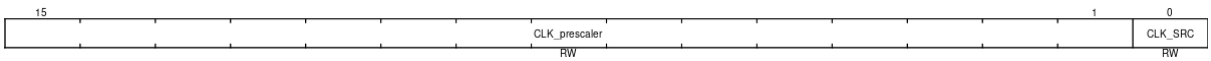


Fig. 6.9.1: PWM Clock control register

Table 6.9.3: Register Bit Fields

Bits	Field Name	Permission	Description
[0:0]	CLK_SRC	RW	Selects PWM clock source: 1 for external, 0 for internal.
[15:1]	CLK_prescaler	RW	Configures clock prescaler to control PWM frequency range.The value must be lesser than 0x5CC8.

CONTROL Register

This 16-bit register is used for enabling/disabling PWM and configuring its mode.It allows enabling/disabling the PWM, setting polarity, starting PWM generation, and handling interrupts.

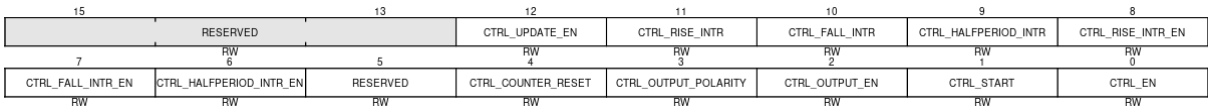


Fig. 6.9.2: PWM Control register

Table 6.9.4: Bit Field Table

Bits	Field Name	Permission	Description
[0:0]	CTRL_EN	RW	If set, PWM is enabled.
[1:1]	CTRL_START	RW	Starts PWM generation when set to 1.
[2:2]	CTRL_OUTPUT_EN	RW	Enables PWM output when set to 1.
[3:3]	CTRL_OUTPUT_POLARITY	RW	When set to 0, the output is inverted (active low). When set to 1, the output is normal (active high).
[4:4]	CTRL_COUNTER_RESET	RW	When set to 1, this bit resets the PWM counter to 0. It should be enabled before disabling any PWM module.
[5:5]	RESERVED	RW	Reserved for future use.
[6:6]	CTRL_HALFPERIOD_INTR_	RW	Enables half-period interrupt when set to 1.
[7:7]	CTRL_FALL_INTR_EN	RW	Enables falling edge interrupt when set to 1.
[8:8]	CTRL_RISE_INTR_EN	RW	Enables rising edge interrupt when set to 1.
[9:9]	CTRL_HALFPERIOD_INTR	RO	Indicates half-period interrupt when set to 1.
[10:10]	CTRL_FALL_INTR	RO	Indicates falling edge interrupt when set to 1.
[11:11]	CTRL_RISE_INTR	RO	Indicates rising edge interrupt when set to 1.
[12:12]	CTRL_UPDATE_EN	RW	Enables/disables immediate register updates when set to 1.
[15:13]	RESERVED	RW	Reserved for future use.

PERIOD Register

This 32-bit register sets the period of the specific pwm instance.The period defines the duration of one complete PWM cycle (inverse of frequency).It can be used to control the switching speed of the PWM signal.

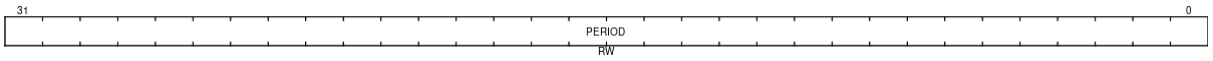


Fig. 6.9.3: PWM Period register

Table 6.9.5: Register Bit Fields

Bits	Field Name	Permission	Description
[31:0]	PERIOD	RW	Specifies the period of the PWM signal, determining the total duration of one PWM cycle.

DUTY CYCLE Register

This 32-bit register is used to set the duty cycle value which controls the percentage of time the PWM signal remains high within a single period.

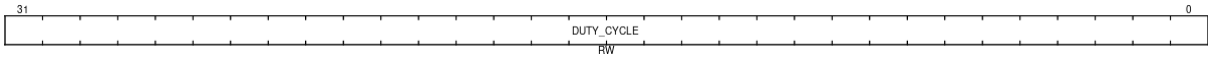


Fig. 6.9.4: PWM Duty cycle register

Table 6.9.6: Register Bit Fields

Bits	Field Name	Permission	Description
[31:0]	DUTY_CYCLE	RW	Sets the PWM duty cycle, controlling the high time ratio and effective voltage.

DEADBAND DELAY Register

This 16-bit register is used to introduce a small delay at rising and falling edges of PWM signals. It sets the delay between rising and falling edges of PWM signals.

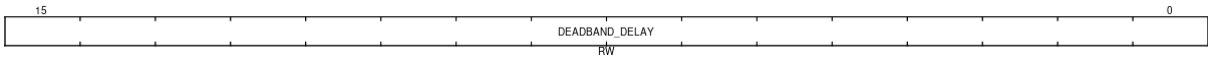
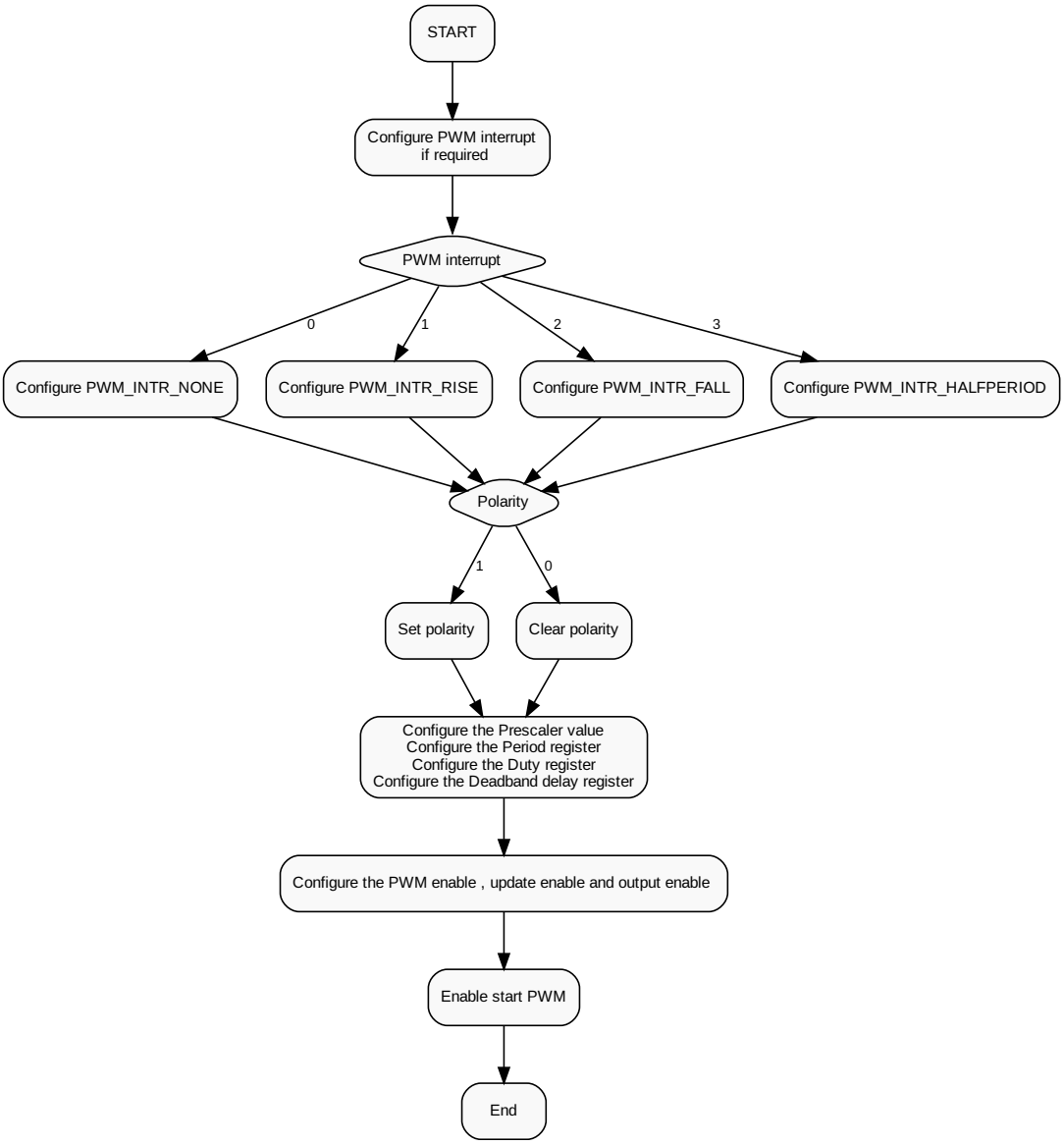


Fig. 6.9.5: PWM Deadband delay register

6.9.3 Workflow

1. Enable PWM pin multiplexing for each selected PWM channel.
2. Configure the PWM clock control register using the selected prescaler value.
3. Program the PWM period, duty cycle, and dead-band delay registers.
4. Configure PWM interrupt sources (rising edge, falling edge, and/or half-period) based on the interrupt mode.
5. Configure PWM output polarity if polarity inversion is enabled.
6. Enable and start PWM operation by setting the control register with update, enable, start, and output-enable bits.
7. Once the output enable is set, the PWM will be generated from the pin.
8. To stop PWM operation, reset the PWM counter for each selected PWM channel.
9. Clear the PWM control, clock, period, duty, and dead-band registers.



6.10 Quad-Serial Peripheral Interface (QSPI)

The Quad Serial Peripheral Interface (QSPI) is an enhanced version of the Serial Peripheral Interface (SPI) protocol designed for high-speed data transfer. It improves upon standard SPI by using four data lines instead of a single data line, significantly increasing data throughput. Secure-IoT devices include two configurable QSPI instances,

QSPI0 and QSPI1, each with independent base addresses and control registers, enabling faster communication with external flash memory devices. QSPI supports high-speed, half-duplex data transfer, making it suitable for high-performance memory applications.

The QSPI supports the following operating modes:

- **Indirect Read Mode:** Used to read data from external memory by explicitly issuing read commands and transferring data through the QSPI FIFO.
- **Indirect Write Mode:** Used to write data to external memory by explicitly issuing write commands and sending data through the QSPI FIFO.
- **Status Polling Mode:** Used to repeatedly read and monitor the status of the external memory until a specified condition is met, without software intervention.
- **XIP (Execute-In-Place) Mode:** Allows code or data to be accessed directly from external flash memory as if it were internal memory.
- **RAM Mode:** Configures the QSPI peripheral to interface with external RAM devices for direct memory access operations.

6.10.1 Instance Details

This section lists the available peripheral instances along with their base addresses and interrupt IDs.

Table 6.10.1: QSPI Instance Register Map

QSPI Instance	Base Address (Hex)	Description
QSPI0	0x00060200	QSPI0
QSPI1	0x00060300	QSPI1

6.10.2 Register Map Details

All registers are 32-bit wide.

Table 6.10.2: QSPI Registers

Register	Offset	Description
CR (Control Register)	0x0000	Used to control various parameters of QSPI such as qspi enable, interrupt enable, setting FIFO threshold, etc.
DCR (Device Configuration Register)	0x0004	Used to configure the clock mode and flash memory size.
SR (Status Register)	0x0008	Status register is a 30-bit register.Holds the status of various flags and FIFO levels.
FCR (Flag Clear Register)	0x000C	Clears the various flags available in QSPI.
DLR (Data Length Register)	0x0010	Used to specify the number of data bytes to be retrieved in indirect and status-polling modes.
CCR (Communication Configuration Register)	0x0014	Set up and manage various aspects of the communication.
AR (Address Register)	0x0018	To store the address of the memory location on an external device.
ABR (Alternate Byte Register)	0x001C	To hold an additional byte of information that may be required by certain external devices, particularly flash memory.
DR (Data Register)	0x0020	Data Register is a 32-bit register used to temporarily hold the data being transmitted to or received from an external device.
PSMKR (Polling Status Mask Register)	0x0028	This register is used to mask off certain bits of the status register during polling.
PSMAR (Polling Status Match Register)	0x002C	this register stores the memory-mapped address used for comparison during a read operation.

continues on next page

Table 6.10.2 – continued from previous page

Register	Offset	Description
PIR (Polling Interval Register)	0x0030	This register stores the polling interval, in clock cycles, used to check QSPI readiness to begin the next operation.
LPTR (Low Power Timeout Register)	0x0034	This 32-bit register is used to store the timeout clock cycles for how long the QSPI should wait for an operation to complete before timing out.
RMC (RAM Mode Configuration Register)	0x0038	This 32-bit register is used to configure the QSPI peripheral during RAM mode.

Control Register (CR)

This 32-bit register contains multiple subfields that determine the type of transaction to be carried out with the device.

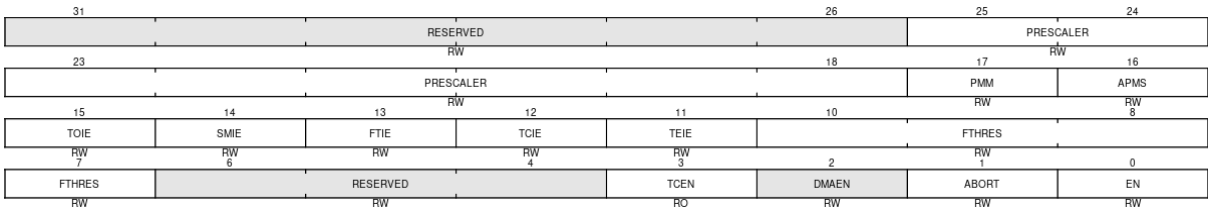


Fig. 6.10.1: QSPI CR

Table 6.10.3: Control Register

Bits	Field Name	Attribute	Description
[0:0]	EN	RW	Enable If set, QSPI communication is enabled.

continues on next page

Table 6.10.3 – continued from previous page

Bits	Field Name	Attribute	Description
[1:1]	ABORT	RW	QSPI Communication Abort request If set, it terminates the ongoing command sequence and automatically resets once the abort process is complete.
[2:2]	DMAEN	RW	DMA transfer enable If set, DMA transfer is enabled.
[3:3]	TCEN	RW	Timeout counter enable This bit enables the timeout counter, which monitors the duration of an ongoing data transfer. If the transfer exceeds the specified timeout period, the command sequence is terminated, and an error flag is set. • **0**: Timeout counter is disabled, allowing the chip select (nCS) to remain active indefinitely after an access in memory-mapped mode. • **1**: Timeout counter is enabled, The timeout counter is activated, causing the chip select to be released in memory-mapped mode after TIMEOUT[15:0] cycles of Flash memory inactivity. Can only be modified when BUSY ~ 0.
[6:4]	RESERVED	RW	Reserved for future use.

continues on next page

Table 6.10.3 – continued from previous page

Bits	Field Name	Attribute	Description
[10:7]	FTHRES	RW	FIFO threshold level In indirect mode, this defines the threshold number of bytes in the FIFO that triggers the FIFO threshold flag to be set.The maximum allowable FTHRES value is 14.When a value “x” is provided, “x + 1” bytes are considered.
[11:11]	TEIE	RW	Transfer error interrupt enable If set, the transfer error interrupt is enabled. This interrupt is generated when a transfer error is detected, such as a timeout or communication failure.
[12:12]	TCIE	RW	Transfer complete interrupt enable If set, the transfer complete interrupt is enabled. This interrupt is generated when the data transfer is successfully completed.
[13:13]	FTIE	RW	FIFO threshold interrupt enable If set, the FIFO threshold interrupt is enabled. This interrupt is generated when the FIFO reaches the configured threshold level during transmission or reception.
[14:14]	SMIE	RW	Status match interrupt enable If set, the status match interrupt is enabled and is triggered when the received data matches the predefined status value in the QSPI status register.

continues on next page

Table 6.10.3 – continued from previous page

Bits	Field Name	Attribute	Description
[15:15]	TOIE	RW	TimeOut interrupt enable If set, the timeout interrupt is enabled and is triggered when the timeout counter expires, indicating that the ongoing command sequence has exceeded the specified time limit.
[16:16]	APMS	RW	Automatic poll mode stop This bit determines if automatic polling is stopped after a match. • **0**: Automatic polling mode is halted through an abort or by disabling the QUADSPI. • **1**: Automatic polling mode stops when there is a match. This bit can be modified only when BUSY ~ 0.

continues on next page

Table 6.10.3 – continued from previous page

Bits	Field Name	Attribute	Description
[17:17]	PMM	RW	Polling match mode This bit indicates which method should be used for determining a “match” during automatic polling mode. • **0**: AND match mode.SMF is set when all the bits received from the Flash memory match the corresponding bits in the match register. • **1**: OR match mode.SMF is set if any of the bits received from the Flash memory matches its corresponding bit in the match register. This bit can only be modified when BUSY ~ 0.
[26:18]	PRESCALER	RW	Clock prescaler This field determines the scaling factor for generating CLK from the interconnect clock.The minimum allowable prescaler value is 6.
[32:27]	RESERVED	RW	Reserved for future use.

Device Configuration Register (DCR)

This is utilized to set the clock mode and define the flash memory size.

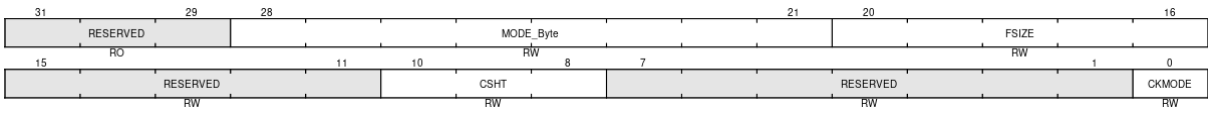


Fig. 6.10.2: QSPI DCR

Table 6.10.4: Device Configuration Register

Bits	Field Name	Attribute	Description
[0:0]	CKMODE	RW	Clock mode This bit indicates the level(High or low) that CLK takes between commands.This can be modified only when BUSY ~ 0.If set, the clock remains high while nCS is high (mode 3). If cleared, the clock remains low while nCS is high (mode 0).
[7:1]	RESERVED	RW	Reserved for future use.
[10:8]	CSHT	RW	Clock mode Chip select high time.
[15:11]	RESERVED	RW	Reserved for future use.
[20:16]	FSIZE	RW	Flash memory size The FSIZE field defines the external memory size in bytes using the formula 2^(FSIZE + 1).
[28:21]	MODE_Byte	RW	Dummy Cycle Mode Byte for Micron Flash
[31:29]	RESERVED	RW	Reserved for future use.

Status Register (SR)

The Status register holds flags to indicate the current state of the QSPI interface. These flags inform about transmission and reception status, error conditions, and FIFO thresholds.

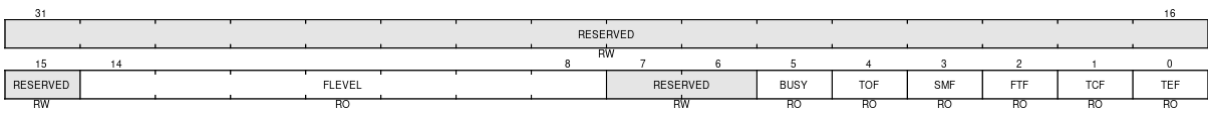


Fig. 6.10.3: QSPI SR

Table 6.10.5: Status Register

Bits	Field Name	Attribute	Description
[0:0]	TEF	RO	Transfer error flag This bit is set in indirect mode when an invalid address is accessed.
[1:1]	TCF	RO	Transfer complete flag This bit is set in indirect mode once the programmed data transfer is complete, or in any mode when the transfer is aborted.
[2:2]	FTF	RO	FIFO threshold flag In indirect mode, this bit is set when the FIFO threshold has been reached, or if there is any data left in the FIFO after reads from the Flash memory are complete and in automatic polling mode this bit is set every time the status register is read.This flag is set when “FTHRES” number of bytes are empty in Indirect write mode or when “FTHRES” number of bytes are filled in Indirect read mode.
[3:3]	SMF	RO	Status match flag This bit is set in automatic polling mode when the received data matches the corresponding bits in the match register.
[4:4]	TOF	RO	Timeout flag Set when timeout occurs.
[5:5]	BUSY	RO	Busy flag Set while an operation is in progress.

continues on next page

Table 6.10.5 – continued from previous page

Bits	Field Name	Attribute	Description
[7:6]	RESERVED	RW	Reserved for future use.
[14:8]	FLEVEL	RO	FIFO level This field indicates the number of valid bytes currently stored in the FIFO.In memory-mapped mode and in automatic status polling mode, FLEVEL is zero.
[31:15]	RESERVED	RW	Reserved for future use.

Flag Clear Register (FCR)

Flag clear register is used to clear the status of the error flags.

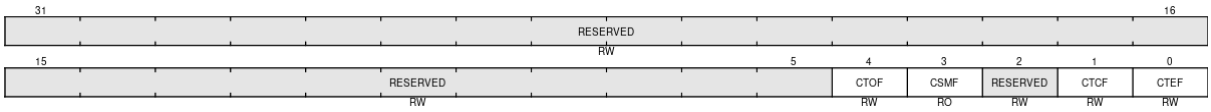


Fig. 6.10.4: QSPI FCR

Table 6.10.6: Flag Clear Register

Bits	Field Name	Attribute	Description
[0:0]	CTEF	WO	Clears Transfer error flag from Status register, when set.
[1:1]	CTCF	WO	Clears Transfer complete flag from Status register, when set.
[2:2]	RESERVED	RW	Reserved for future use.
[3:3]	CSMF	WO	Clears Status match flag from Status register, when set.
[4:4]	CTOF	WO	Clears Timeout flag from Status register, when set.

continues on next page

Table 6.10.6 – continued from previous page

Bits	Field Name	Attribute	Description
[31:5]	RESERVED	RW	Reserved for future use.

Data Length Register (DLR)

This register specifies the number of data bytes to be retrieved in indirect and status-polling modes. This register has RW access.

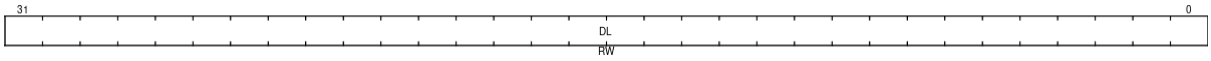


Fig. 6.10.5: QSPI DLR

Communication Configuration Register (CCR)

This 32-bit register is used to configure the mode and address size of the QSPI transaction.The address mode can be configured as Nil, single line, two line or four line.The address size can be configured as 8, 16, 24, or 32 bit.

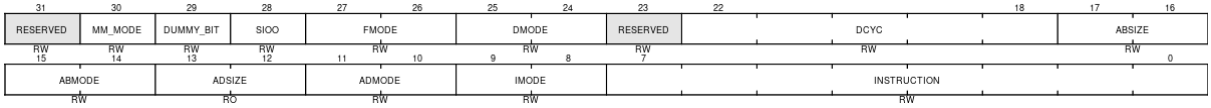


Fig. 6.10.6: QSPI CCR

Note

The below fields can be written only when BUSY~0.

Table 6.10.7: Communication Configuration Register

Bits	Field Name	Attribute	Description
[7:0]	INSTRUCTION	RW	The Instruction to be transmitted to the external device.

continues on next page

Table 6.10.7 – continued from previous page

Bits	Field Name	Attribute	Description
[9:8]	IMODE	RW	Instruction mode This specifies the instruction phase mode of operation. • **00**: No instruction • **01**: Single line instruction • **10**: Two lines instruction • **11**: Four lines instruction
[11:10]	ADMODE	RW	Address mode Specifies the address phase mode of operation. • **00**: No address • **01**: Address on single line • **10**: Address on two lines • **11**: Address on four lines
[13:12]	ADSIZE	RW	Address Size This bit defines address size. • **00**: 8-bit address • **01**: 16-bit address • **10**: 24-bit address • **11**: 32-bit address

continues on next page

Table 6.10.7 – continued from previous page

Bits	Field Name	Attribute	Description
[15:14]	ABMODE	RW	Alternate byte mode Specifies the alternate-bytes phase mode of operation. • **00**: No alternate byte • **01**: Alternate byte in single line • **10**: Alternate byte two lines • **11**: Alternate byte four lines
[17:16]	ABSIZE	RW	Alternate byte size Specifies the size of Alternate byte • **00**: 8-bit • **01**: 16-bit • **10**: 24-bit • **11**: 32-bit
[22:18]	DCYC	RW	Dummy cycles Specifies the duration of the dummy phase. In both SDR and DDR modes, it specifies a number of dummy CLK cycles.
[23:23]	RESERVED	RW	Reserved for future use.

continues on next page

Table 6.10.7 – continued from previous page

Bits	Field Name	Attribute	Description
[25:24]	DMODE	RW	Data mode This bit selects the data mode for the QSPI communication. It determines how the data will be transmitted, such as single, dual, or quad I/O mode. • **00**: No data • **01**: Single line transaction • **10**: Two lines transaction • **11**: Four lines transaction
[27:26]	FMODE	RW	Functional mode - Specifies the functional mode of operation. • **00**: Indirect write mode • **01**: Indirect read mode • **10**: Automatic polling mode • **11**: Memory-mapped mode

continues on next page

Table 6.10.7 – continued from previous page

Bits	Field Name	Attribute	Description
[28:28]	SIOO	RW	Send instruction only once mode This bit enables the Send Instruction Only Once mode, allowing the instruction to be sent just once at the start of the command sequence. • **0**: Send instruction on every transaction. • **1**: Send instruction only for the first command.
[29:29]	DUMMY_BIT	RW	Dummy bit Set value 1 to send Dummy Cycles.Default value of this field is 0.
[30:30]	MM_MODE	RW	Memory Mapped Mode Sets the Memory-mapped mode.Default value 0. • **0**: XIP mode • **1**: RAM mode
[31:31]	RESERVED	RW	Reserved for future use.

Address Register (AR)

This 32-bit data is used to store the address of the external device such as Flash memory from which data is to be read. It has RW access.

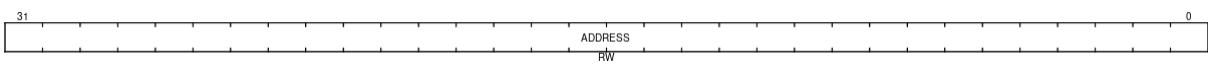


Fig. 6.10.7: QSPI AR

Alternate Byte Register (ABR)

This register is used to specify an additional byte or sequence of bytes that can be sent during communication. This field is writable only when BUSY ~ 0. This bit has RW access.

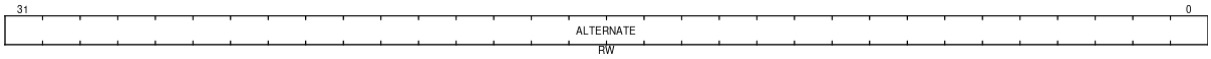


Fig. 6.10.8: QSPI ABR

Data Register (DR)

This register stores the data to be transmitted during write operations and the data received from the external QSPI device during read operations. This register has RW access.

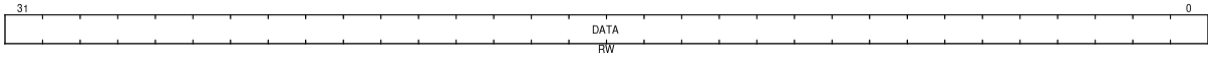


Fig. 6.10.9: QSPI DR

Polling Status Mask Register (PSMKR)

This register stores the mask value used during status polling mode. This register has RW access.The PSMKR register is used to mask off certain bits of the status register during polling.

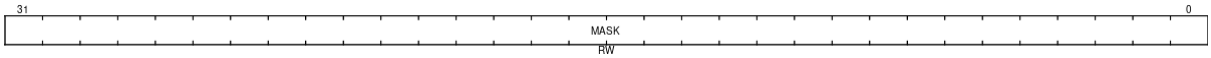


Fig. 6.10.10: PSMKR

Polling Status Match Register (PSMAR)

This register stores the Match value used during status polling mode. This register is used to specify the memory-mapped address that the QSPI peripheral will compare against during a read operation. This register has RW access.

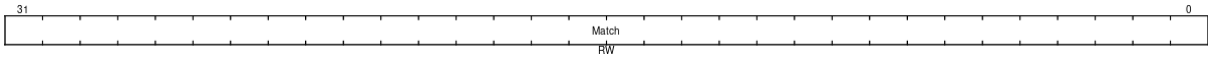


Fig. 6.10.11: PSMAR

Polling Interval Register (PIR)

The poll interval is a 32-bit register that represents the time interval (in clock cycles) between two consecutive polls of the QSPI peripheral to check if it is ready for a new operation.

Fig. 6.10.12: QSPI PIR

Low power timeout Register (LPTR)

This register stores the timeout value for the QSPI peripheral in clock cycles. This value is used to determine how long the QSPI peripheral should wait for an operation to complete before timing out. This register has RW access.

Fig. 6.10.13: QSPI LPTR

RAM mode configuration Register (RMC)

This register holds the configuration details for RAM mode.

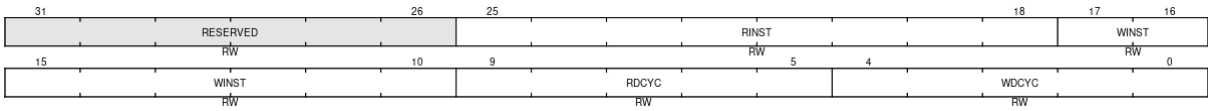


Fig. 6.10.14: QSPI RMC

Table 6.10.8: RAM mode configuration Register

Bits	Field Name	Attribute	Description
[0:0]	WDCYC	WO	Clears Transfer error flag from Status register, when set.
[1:1]	CTCF	WO	Clears Transfer complete flag from Status register, when set.
[2:2]	RESERVED	RW	Reserved for future use.
[3:3]	CSMF	WO	Clears Status match flag from Status register, when set.
[4:4]	CTOF	WO	Clears Timeout flag from Status register, when set.
[31:5]	RESERVED	RW	Reserved for future use.

6.10.3 Workflow

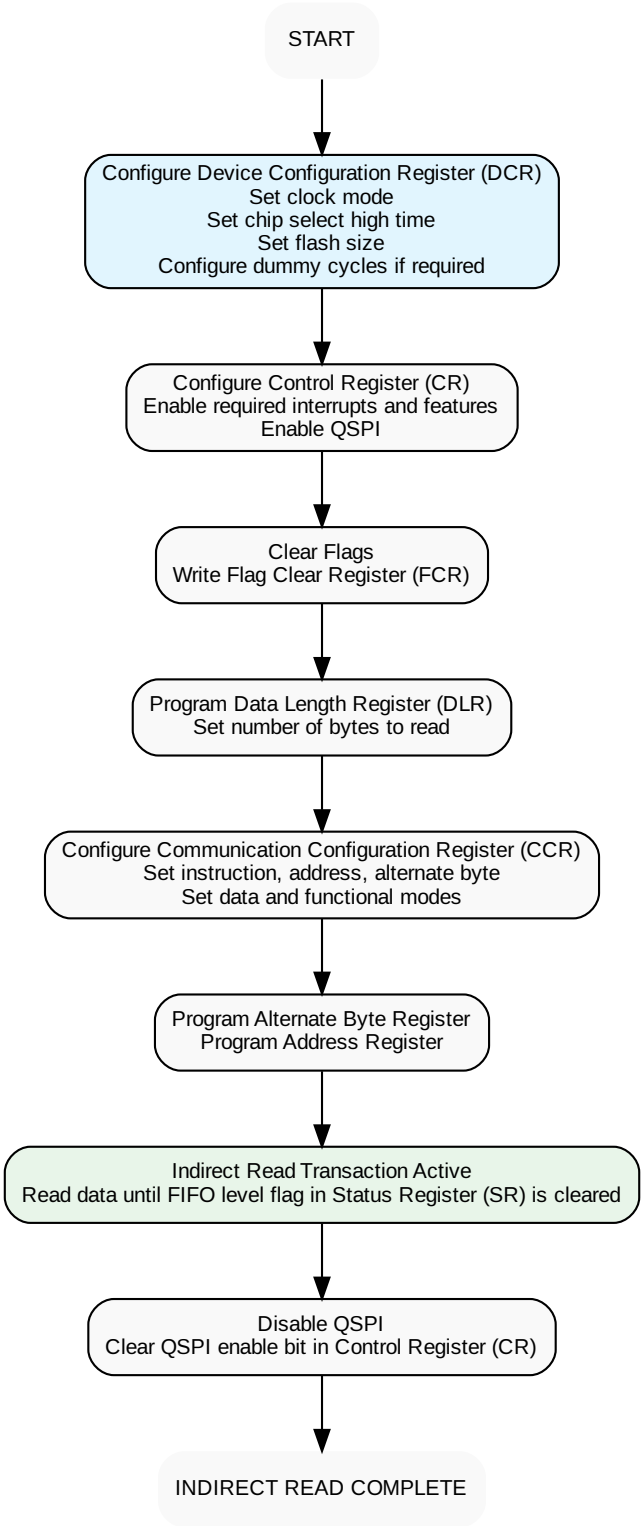
Note

Ensure that the BUSY bit in the Status Register (SR) is cleared before configuring any register or performing read or write operations.

Indirect Read mode

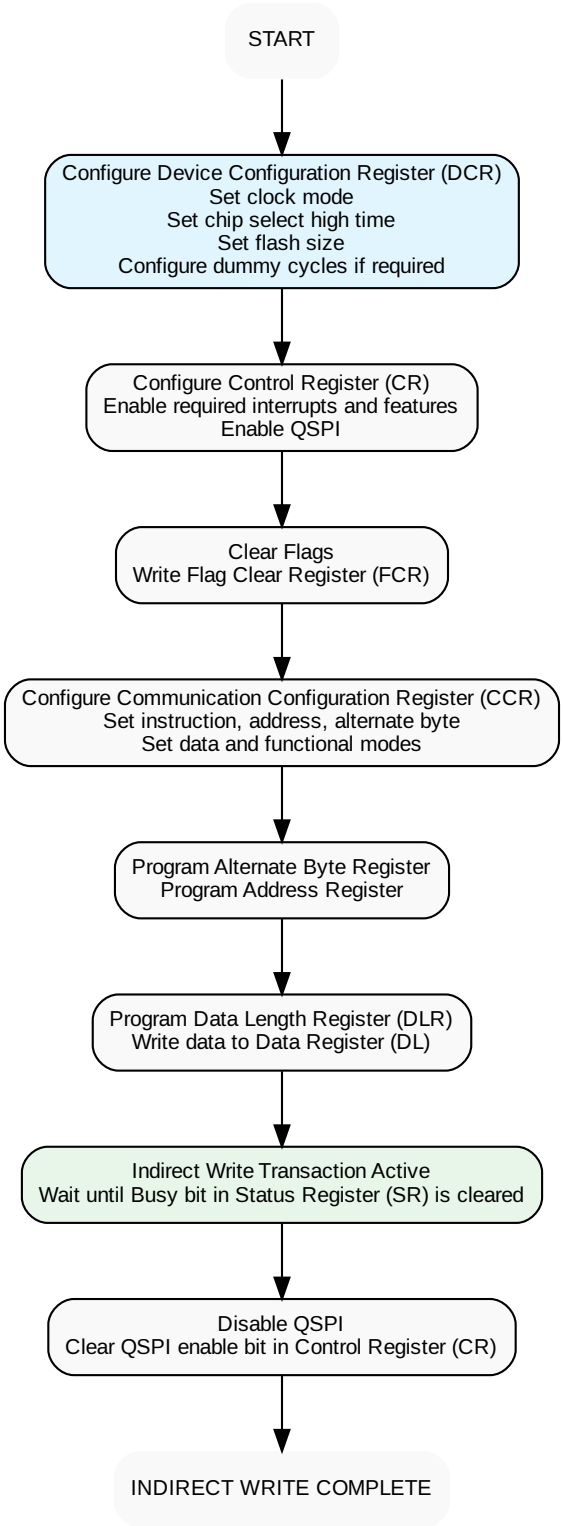
1. Configure the clock mode, chip-select high time, flash memory size, and the dummy cycle mode byte (if required) in the Device Configuration Register (DCR).
2. Enable the required interrupts and features, and set the QSPI enable bit in the Control Register (CR).
3. Clear unnecessary flags by writing to the Flag Clear Register (FCR).
4. Program the Data Length Register (DLR) with the number of bytes to be read.
5. Configure the instruction, address, alternate byte, data, and functional modes, along with other required features, in the Communication Configuration Register (CCR).
6. Program the Alternate Byte Register(ABR) and the Address Register (AR).

7. Start the transaction and read data until the FIFO level flag in the Status Register (SR) is cleared.
8. Clear the enable bit in the Control Register (CR) to disable the QSPI.


START
Configure Device Configuration Register (DCR)
Set clock mode
Set chip select high time
Set flash size
Configure dummy cycles if required
Configure Control Register (CR)
Enable required interrupts and features
Enable QSPI
Clear Flags
Write Flag Clear Register (FCR)
Program Data Length Register (DLR)
Set number of bytes to read
Configure Communication Configuration Register (CCR)
Set instruction, address, alternate byte
Set data and functional modes
Program Alternate Byte Register
Program Address Register
Indirect Read Transaction Active
Read data until FIFO level flag in Status Register (SR) is cleared
Disable QSPI
Clear QSPI enable bit in Control Register (CR)
INDIRECT READ COMPLETE

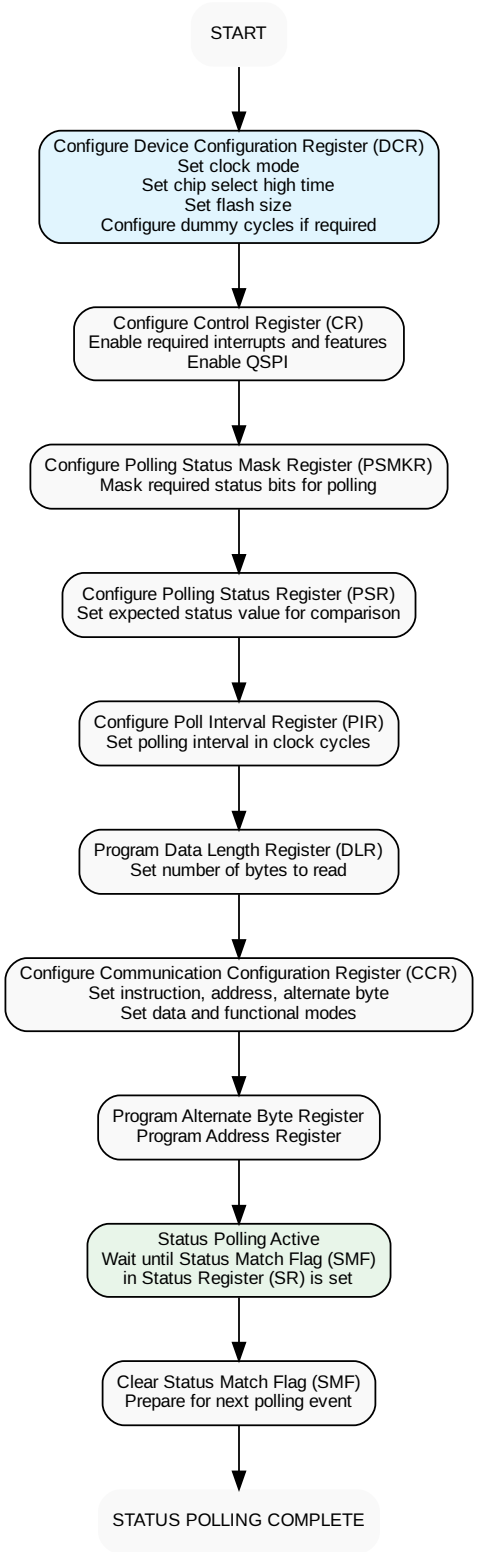
Indirect Write mode

1. Configure the clock mode, chip select high time, Flash memory size and if dummy cycle mode byte if needed in the Device Configuration Register(DCR).
2. Enable the required interrupts, features and enable the QSPI EN in the Control register(CR).
3. Clear the unnecessary flags in Flag clear register (FCR).
4. Configure the instruction, address, alternate byte, data and function modes along with others features in the Communication Configuration Register(CCR).
5. Update the Alternate byte register (ABR) and Address register(AR).
6. Program the length of data tp the Data Length Register (DLR) and the data to the Data Register(DL).
7. Start the transaction and wait until the Busy bit in the Status Register (SR) is cleared.
8. Clear the enable bit in the Control register(CR) to disable the QSPI.


START
Configure Device Configuration Register (DCR)
Set clock mode
Set chip select high time
Set flash size
Configure dummy cycles if required
Configure Control Register (CR)
Enable required interrupts and features
Enable QSPI
Clear Flags
Write Flag Clear Register (FCR)
Configure Communication Configuration Register (CCR)
Set instruction, address, alternate byte
Set data and functional modes
Program Alternate Byte Register
Program Address Register
Program Data Length Register (DLR)
Write data to Data Register (DL)
Indirect Write Transaction Active
Wait until Busy bit in Status Register (SR) is cleared
Disable QSPI
Clear QSPI enable bit in Control Register (CR)
INDIRECT WRITE COMPLETE

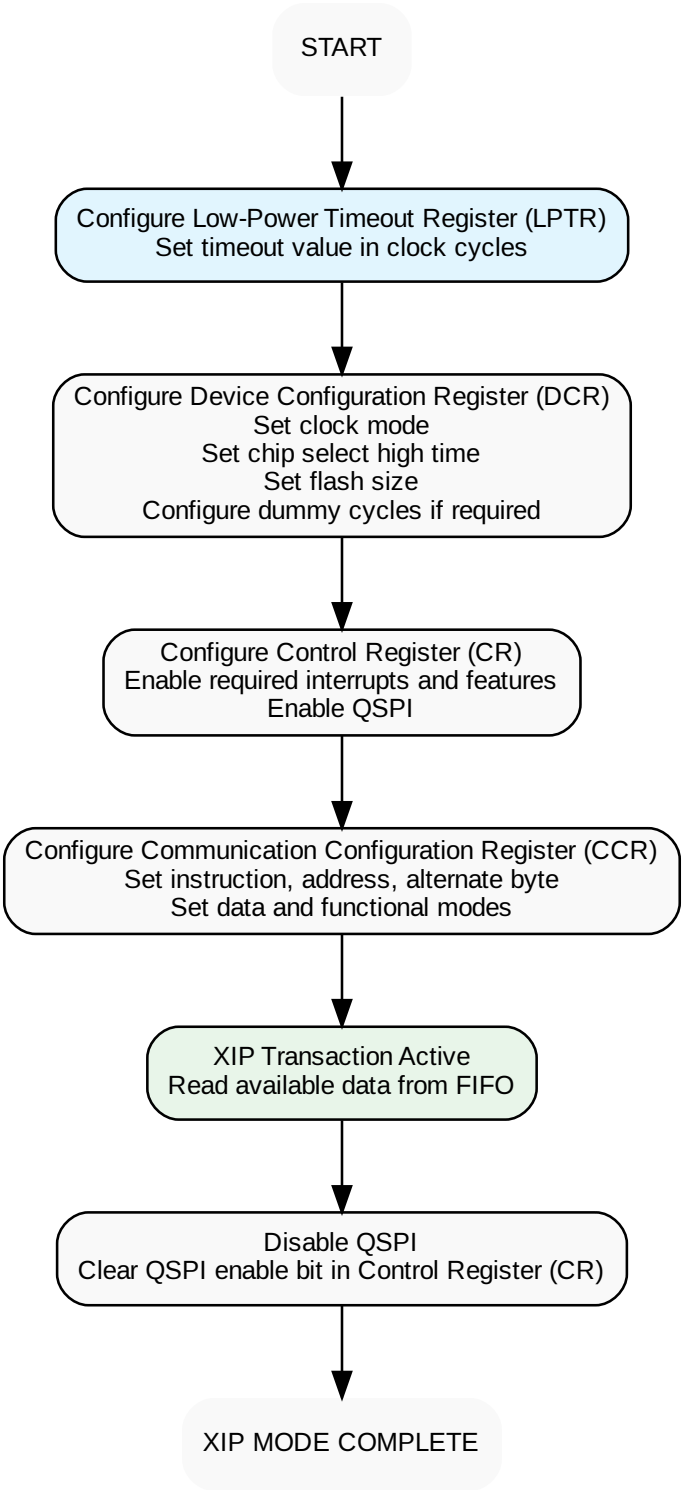
Status polling mode

1. Configure the clock mode, chip select high time, Flash memory size and if dummy cycle mode byte if needed in the Device Configuration Register(DCR).
2. Enable the required interrupts, features and enable the QSPI EN in the Control register(CR).
3. Configure the mask value in the PSMKR register to mask the required bits in the status register for polling.
4. Check the status register that the QSPI peripheral will compare against during a read operation.
5. Configure the Poll interval register (PIR), with the poll interval value (in clock cycles).
6. Program the length of data in the Data Length Register (DLR).
7. Configure the instruction, address, alternate byte, data and function modes along with others features in the Communication Configuration Register(CCR).
8. Program the Aternate byte register and address register.
9. Wait till the Status match flag (SMF) bit of the SR is set.
10. Once the SMF is set, clear the flag to detect the next occurance.


START
Configure Device Configuration Register (DCR)
Set clock mode
Set chip select high time
Set flash size
Configure dummy cycles if required
Configure Control Register (CR)
Enable required interrupts and features
Enable QSPI
Configure Polling Status Mask Register (PSMKR)
Mask required status bits for polling
Configure Polling Status Register (PSR)
Set expected status value for comparison
Configure Poll Interval Register (PIR)
Set polling interval in clock cycles
Program Data Length Register (DLR)
Set number of bytes to read
Configure Communication Configuration Register (CCR)
Set instruction, address, alternate byte
Set data and functional modes
Program Alternate Byte Register
Program Address Register
Status Polling Active
Wait until Status Match Flag (SMF)
in Status Register (SR) is set
Clear Status Match Flag (SMF)
Prepare for next polling event
STATUS POLLING COMPLETE

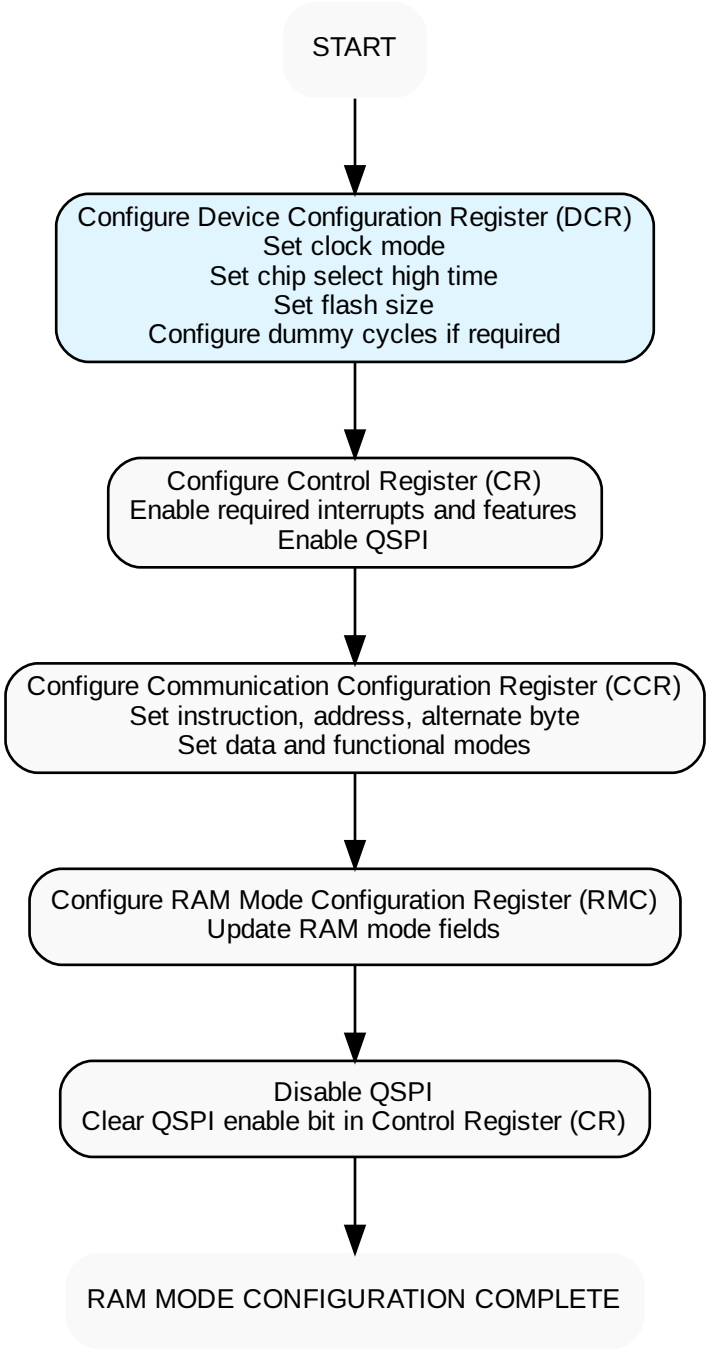
XIP mode

1. Update the LPTR with the number of clock cycles the QSPI peripheral should wait for an operation to complete before timing out.
2. Configure the clock mode, chip select high time, Flash memory size and if dummy cycle mode byte if needed in the Device Configuration Register(DCR).
3. Enable the required interrupts, features and enable the QSPI EN in the Control register(CR).
4. Configure the instruction, address, alternate byte, data and function modes along with others features in the Communication Configuration Register(CCR).
5. Read the available data from the FIFO.
6. Clear the enable bit in the Control register to disable the QSPI.


START
Configure Low-Power Timeout Register (LPTR)
Set timeout value in clock cycles
Configure Device Configuration Register (DCR)
Set clock mode
Set chip select high time
Set flash size
Configure dummy cycles if required
Configure Control Register (CR)
Enable required interrupts and features
Enable QSPI
Configure Communication Configuration Register (CCR)
Set instruction, address, alternate byte
Set data and functional modes
XIP Transaction Active
Read available data from FIFO
Disable QSPI
Clear QSPI enable bit in Control Register (CR)
XIP MODE COMPLETE

RAM mode

1. Configure the clock mode, chip select high time, Flash memory size and if dummy cycle mode byte if needed in the Device Configuration Register(DCR).
2. Enable the required interrupts, features and enable the QSPI EN in the Control register(CR).
3. Configure the instruction, address, alternate byte, data and function modes along with others features in the Communication Configuration Register(CCR).
4. Configure the fields in the RAM Mode configuration register (RMC).
5. Clear the enable bit in the Control register to disable the QSPI.


START
Configure Device Configuration Register (DCR)
Set clock mode
Set chip select high time
Set flash size
Configure dummy cycles if required
Configure Control Register (CR)
Enable required interrupts and features
Enable QSPI
Configure Communication Configuration Register (CCR)
Set instruction, address, alternate byte
Set data and functional modes
Configure RAM Mode Configuration Register (RMC)
Update RAM mode fields
Disable QSPI
Clear QSPI enable bit in Control Register (CR)
RAM MODE CONFIGURATION COMPLETE

6.11 SPI (Serial Peripheral Interface)

SPI is a fast synchronous serial protocol used for short-distance communication between a master and slave.This sections deals with SPI peripheral in S2401 . The SPI peripheral can be configured as both Master and Slave. Each instance has interrupts registered with PLIC.

Note

The SPI controller only supports **Mode 0** and **Mode 3**. The SPI slave does not support half-duplex mode. In full-duplex and TX mode, the SPI slave sends a redundant **0**; the master must perform a redundant read as its first operation.

6.11.1 SPI Instance details

Table 6.11.1: SPI instance details

SPI Instance	Base Address	Interrupt ID
SPI0	0x20000	62
SPI1	0x20100	63
SPI2	0x20200	64
SPI3	0x20300	65

Note

SPI1 and SPI3 are pinmuxed For detailed information on SPI pinmux, refer to *Pinmux Register Map*.

6.11.2 SPI Register Map

The register map for SPI peripheral is listed below.

Table 6.11.2: Register Map

Register Name	Offset	Size (Bits)	Description
CTRL (Communication Control Register)	0x00	32	SPI communication control register.
CLK_CTRL (clock control Register)	0x04	32	SPI clock generation control register.
SPI_TX (Transmit register)	0x08	32	Holds the tx data.
SPI_RX (Receive Register)	0x0c	32	Holds the rx data.
SPI_INTR_EN (Interrupt Enable Register)	0x10	32	SPI Interrupt enable register.
SPI_FIFO_STATUS (FIFO status Register)	0x14	32	Gives the status of TX/RX FIFO.
SPI_COMM_STATUS (Communication status register)	0x18	16	SPI communication status register.
NCS_CTRL(Chip select control register)	0x1C	8	Chip select control Register

6.11.3 Register details

CTRL (Communication Control Register)

The **CTRL** is a 32-bit register used to configure spi mode(ie. Master, Slave), comm_mode (i.e. TX, RX, Full-duplex, half-duplex), total number of bits to be transmitted or received, whether the transmission should be lsb first or msb first and options to flush the RX FIFO. It also allows enabling or disabling MISO, MOSI, NCS, and SCLK outen, as well as configuring the DMA transaction size.

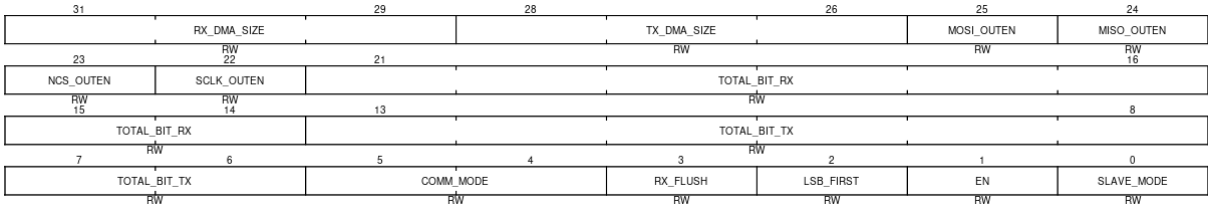


Fig. 6.11.1: SPI Communication Control Register

Table 6.11.3: CTRL Register

Bits	Field Name	Permission	Description
[0:0]	SLAVE_MOI	RW	If set, SPI acts as a slave device; otherwise it operates in master mode.
[1:1]	EN	RW	When set, the SPI transaction starts.
[2:2]	LSB_FIRST	RW	If set, data is transmitted LSB first; otherwise MSB first.
[3:3]	RX_FLUSH	RW	Set to flush the RX FIFO.
[5:4]	COMM_MOI	RW	Communication modes of SPI: • **00** – TX • **01** – RX • **10** – Half-duplex (transmit then immediate receive) • **11** – Full-duplex (simultaneous transmit and receive)
[13:6]	TOTAL_BIT_TX	RW	Total number of bits to be transmitted in an SPI transaction.
[21:14]	TOTAL_BIT_RX	RW	Total number of bits to be received in an SPI transaction.
[22:22]	SCLK_OUTE	RW	Enables SCLK output. If set, the controller generates SCLK; used for master mode.
[23:23]	NCS_OUTEN	RW	Enables NCS output. If set, the controller generates NCS; used for master mode.

continues on next page

Table 6.11.3 – continued from previous page

Bits	Field Name	Permis-sion	Description
[24:24]	MISO_OUTE	RW	Enables MISO output. If set, data is driven on MISO; used for slave mode.
[25:25]	MOSI_OUTE	RW	Enables MOSI output. If set, data is driven on MOSI; used for master mode.
[28:26]	TX_DMA_SI	RW	DMA transmit transaction size in bytes.
[31:29]	RX_DMA_SI	RW	DMA receive transaction size in bytes.

CLK_CTRL (clock control Register)

The **CLK_CTRL** is a 32-bit register used to configure the spi clock. It can set the polarity, phase, prescalar of the clock.

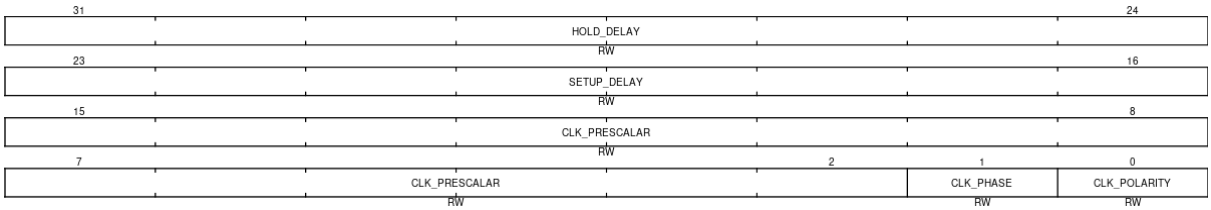


Fig. 6.11.2: SPI clock Control Register

Table 6.11.4: CLK_CTRL Register

Bits	Field Name	Permission	Description
[0:0]	CLK_POLARITY	RW	Holds the clock polarity.
[1:1]	CLK_PHASE	RW	Holds the clock phase.
[15:2]	CLK_PRESCALAR	RW	Holds the prescaler value of the sclk.
[23:16]	SETUP_DELAY	RW	Holds the setup delay.
[31:24]	HOLD_DELAY	RW	Holds the hold delay.

Note

S2401 SPI only supports the following modes: mode 0 {CPOL - 0, CPHA - 0} and mode 1 {CPOL - 1, CPHA - 1}.

SPI_TX (Transmit register)

The **SPI_TX** register is a 32-bit register, Which holds the data to be transmitted. Data written here is transferred to the TX FIFO

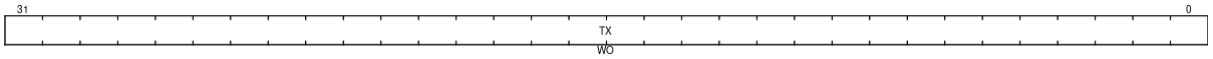


Fig. 6.11.3: SPI Transmit Register

Note

Software must check the respective flag in the *SPI_FIFO_STATUS (FIFO status Register)* register before writing ; attempting to write to TX register when TX FIFO is full will result in data loss.

SPI_RX (Receive Register)

The SPI_RX is a 32-bit read-only register that retrieves the available data from the RX FIFO.

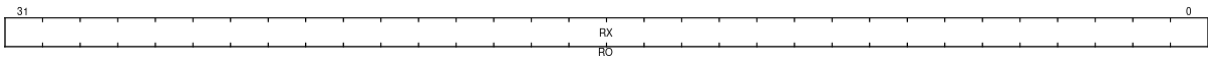


Fig. 6.11.4: SPI Receive Register

Note

Software must check the respective flag in the *SPI_FIFO_STATUS (FIFO status Register)* register before reading ; attempting to read RX register when RX FIFO is emepty will result in garbage value.

SPI_INTR_EN (Interrupt Enable Register)

The **SPI_INTR_EN** is a 32-bit register. The following table describes the flags in the 32-bit Interrupt enable register with their corresponding bits, field names, permissions and description.

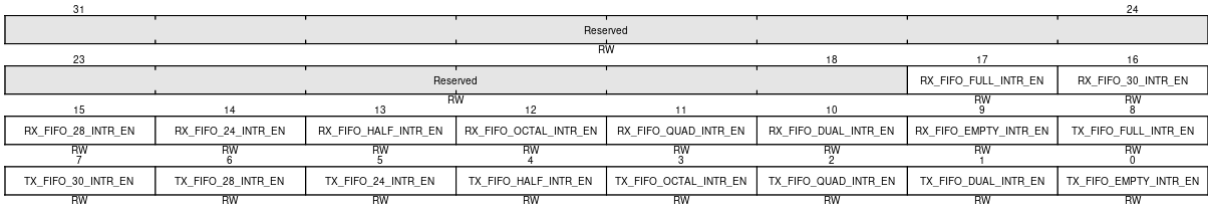


Fig. 6.11.5: SPI Interrupt Enable Register

Table 6.11.5: SPI_INTR_EN Register Details

Bits	Field Name	Permis-sion	Description
[0:0]	TX_FIFO_EMPTY_IN	RW	If set, interrupt is sent to PLIC when TX FIFO is empty.
[1:1]	TX_FIFO_DUAL_INTF	RW	If set, interrupt is sent to PLIC when TX FIFO is filled by 2 entries.
[2:2]	TX_FIFO_QUAD_INTF	RW	If set, interrupt is sent to PLIC when TX FIFO is filled by 4 entries.
[3:3]	TX_FIFO_OCTAL_INT	RW	If set, interrupt is sent to PLIC when TX FIFO is filled by 8 entries.
[4:4]	TX_FIFO_HALF_INTF	RW	If set, interrupt is sent to PLIC when TX FIFO is filled by 16 entries.
[5:5]	TX_FIFO_24_INTR_EI	RW	If set, interrupt is sent to PLIC when TX FIFO is filled by 24 entries.
[6:6]	TX_FIFO_28_INTR_EI	RW	If set, interrupt is sent to PLIC when TX FIFO is filled by 28 entries.
[7:7]	TX_FIFO_30_INTR_EI	RW	If set, interrupt is sent to PLIC when TX FIFO is filled by 30 entries.
[8:8]	TX_FIFO_FULL_INTR	RW	If set, interrupt is sent to PLIC when TX FIFO is full - 32 entries.
[9:9]	RX_FIFO_EMPTY_IN	RW	If set, interrupt is sent to PLIC when RX FIFO is empty.

continues on next page

Table 6.11.5 – continued from previous page

Bits	Field Name	Permis-sion	Description
[10:10]	RX_FIFO_DUAL_INTF	RW	If set, interrupt is sent to PLIC when RX FIFO is filled by 2 entries.
[11:11]	RX_FIFO_QUAD_INTF	RW	If set, interrupt is sent to PLIC when RX FIFO is filled by 4 entries.
[12:12]	RX_FIFO_OCTAL_INT	RW	If set, interrupt is sent to PLIC when RX FIFO is filled by 8 entries.
[13:13]	RX_FIFO_HALF_INTF	RW	If set, interruptis sent to PLIC when RX FIFO is filled by 16 entries.
[14:14]	RX_FIFO_24_INTR_E	RW	If set, interrupt is sent to PLIC when RX FIFO is filled by 24 entries.
[15:15]	RX_FIFO_28_INTR_E	RW	If set, interrupt is sent to PLIC when RX FIFO is filled by 28 entries.
[16:16]	RX_FIFO_30_INTR_E	RW	If set, interrupt is sent to PLIC when RX FIFO is filled by 30 entries.
[17:17]	RX_FIFO_FULL_INTR	RW	If set, interrupt is sent to PLIC when RX FIFO is full - 32 entries.
[31:18]	Reserved	RW	Reserved bit for future use.

SPI_FIFO_STATUS (FIFO status Register)

The **SPI_FIFO_STATUS** is a 32-bit register that holds flags to indicate the current state of the SPI interface. These flags provide information about the TX FIFO and RX FIFO. TX and RX FIFO's can hold 32 bytes.

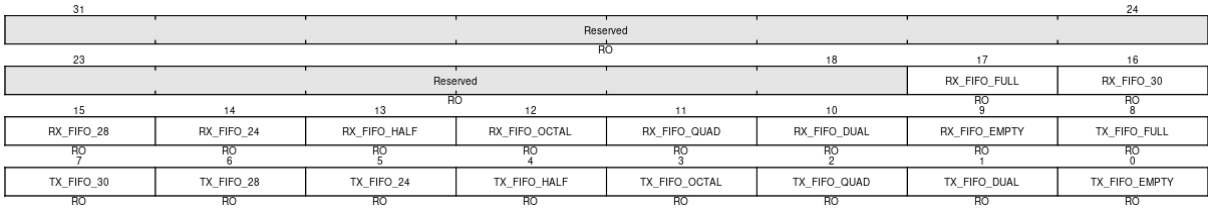


Fig. 6.11.6: SPI FIFO status Register

Table 6.11.6: SPI_FIFO_STATUS Register Details

Bits	Field Name	Permission	Description
[0:0]	TX_FIFO_EMPTY	RO	Set when TX FIFO is empty.
[1:1]	TX_FIFO_DUAL	RO	Set when TX FIFO is filled by 2 entries.
[2:2]	TX_FIFO_QUAD	RO	Set when TX FIFO is filled by 4 entries.
[3:3]	TX_FIFO_OCTAL	RO	Set when TX FIFO is filled by 8 entries.
[4:4]	TX_FIFO_HALF	RO	Set when TX FIFO is filled by 16 entries.
[5:5]	TX_FIFO_24	RO	Set when TX FIFO is filled by 24 entries.
[6:6]	TX_FIFO_28	RO	Set when TX FIFO is filled by 28 entries.
[7:7]	TX_FIFO_30	RO	Set when TX FIFO is filled by 30 entries.
[8:8]	TX_FIFO_FULL	RO	Set when TX FIFO is full - 32 entries.
[9:9]	RX_FIFO_EMPTY	RO	Set when RX FIFO is empty.
[10:10]	RX_FIFO_DUAL	RO	Set when RX FIFO is filled by 2 entries.
[11:11]	RX_FIFO_QUAD	RO	Set when RX FIFO is filled by 4 entries.
[12:12]	RX_FIFO_OCTAL	RO	Set when RX FIFO is filled by 8 entries.
[13:13]	RX_FIFO_HALF	RO	Set when RX FIFO is filled by 16 entries.
[14:14]	RX_FIFO_24	RO	Set when RX FIFO is filled by 24 entries.

continues on next page

Table 6.11.6 – continued from previous page

Bits	Field Name	Permission	Description
[15:15]	RX_FIFO_28	RO	Set when RX FIFO is filled by 28 entries.
[16:16]	RX_FIFO_30	RO	Set when RX FIFO is filled by 30 entries.
[17:17]	RX_FIFO_FULL	RO	Set when RX FIFO is full - 32 entries.
[31:18]	Reserved	RW	Reserved bit for future use.

SPI_COMM_STATUS (Communication status register)

The **SPI_COMM_STATUS** is a 16-bit register that provide information about whether the Transmission or reception is started via spi interface and FIFO depths.

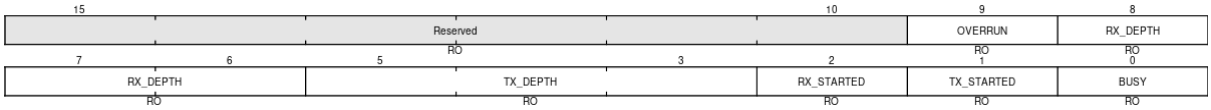


Fig. 6.11.7: SPI Communication status register

Table 6.11.7: SPI_COMM_STATUS Register

Bits	Field Name	Permission	Description
[0:0]	BUSY	RO	This will be set when NCS goes low and will be reset when NCS goes high.
[1:1]	TX_STARTEI	RO	Set on transmit starts and Cleared on transmit completion.
[2:2]	RX_STARTEI	RO	Cleared on receive start; set on receive completion.

continues on next page

Table 6.11.7 – continued from previous page

Bits	Field Name	Permis-sion	Description
[5:3]	TX_DEPTH	RO	The TX_DEPTH field is 3 bits wide and specifies the number of entries in the TX FIFO. • **000**: TX FIFO contains 0 to 1 entry. • **001**: TX FIFO contains 2 to 3 entries. • **010**: TX FIFO contains 4 to 7 entries. • **011**: TX FIFO contains 8 to 15 entries. • **100**: TX FIFO contains 16 to 23 entries. • **101**: TX FIFO contains 24 to 27 entries. • **110**: TX FIFO contains 28 to 29 entries. • **111**: TX FIFO contains 30 to 31 entries.
[8:6]	RX_DEPTH	RO	The RX_DEPTH field is 3 bits wide and specifies the number of entries in the RX FIFO. • **000**: RX FIFO contains 0 to 1 entry. • **001**: RX FIFO contains 2 to 3 entries. • **010**: RX FIFO contains 4 to 7 entries. • **011**: RX FIFO contains 8 to 15 entries. • **100**: RX FIFO contains 16 to 23 entries. • **101**: RX FIFO contains 24 to 27 entries. • **110**: RX FIFO contains 28 to 29 entries. • **111**: RX FIFO contains 30 to 31 entries.
[9:9]	OVERRUN	RO	This will be set when there is an overrun during receive operation.
[15:10]	Reserved	RW	Reserved bit for future use.

NCS_CTRL(Chip select control register)

The **NCS_CTRL** is a 8-bit register used to select between the hardware ncs and the software ncs.

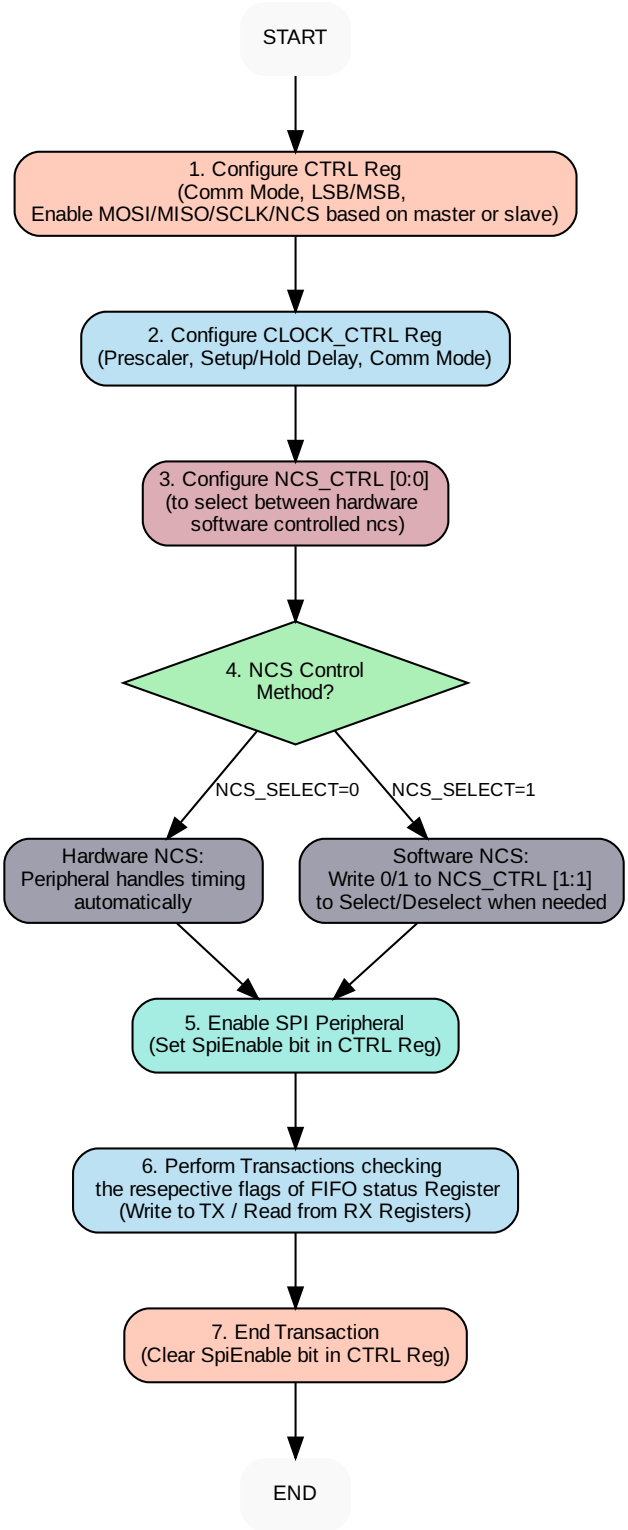
Fig. 6.11.8: SPI Chip select control register

Table 6.11.8: NCS_CTRL Register

Bits	Field Name	Permission	Description
[0:0]	NCS_SELEC	RW	If set,uses the software controlled ncs. Else , uses the hardware ncs .
[1:1]	NCS_SW	RW	Holds the value of software controlled ncs .If set ,high (deselect) else low (select).
[7:2]	RESERVED	RW	Reserved for future use

6.11.4 Workflow

1. Primary Configuration: Set the CTRL Register for Master/Slave role, bit order (LSB/MSB), and enable the MOSI, MISO, SCLK, and NCS pins.
2. Clock Setup: Write to the CLOCK_CTRL Register to set the clock prescaler and define setup/hold delays.
3. NCS Mode Selection: Configure NCS_CTRL [0:0] to choose between Hardware NCS (0) or Software NCS (1).
4. NCS Initiation:
   - Hardware Path: The peripheral handles timing automatically.
   - Software Path: Write 0/1 to NCS_CTRL [1:1] to Select/Deselect when needed
5. Enable Peripheral: Set the SpiEnable bit in the CTRL Register to start the SPI engine.
6. Data Transaction: Perform reads/writes while polling the FIFO_STATUS Register:
7. Termination: Clear the SpiEnable bit in the CTRL Register. If using Software NCS, write 1 to NCS_CTRL [1:1] to deselect the peripheral.


START
1. Configure CTRL Reg
(Comm Mode, LSB/MSB,
Enable MOSI/MISO/SCLK/NCS based on master or slave)
2. Configure CLOCK_CTRL Reg
(Prescaler, Setup/Hold Delay, Comm Mode)
3. Configure NCS_CTRL [0:0]
(to select between hardware
software controlled ncs)
4. NCS Control
Method?
NCS_SELECT=0
NCS_SELECT=1
Hardware NCS:
Peripheral handles timing
automatically
Software NCS:
Write 0/1 to NCS_CTRL [1:1]
to Select/Deselect when needed
5. Enable SPI Peripheral
(Set SpiEnable bit in CTRL Reg)
6. Perform Transactions checking
the resepective flags of FIFO status Register
(Write to TX / Read from RX Registers)
7. End Transaction
(Clear SpiEnable bit in CTRL Reg)
END

6.12 Universal Asynchronous Receiver-Transmitter (UART)

The Universal Asynchronous Receiver-Transmitter (UART) is a serial communication protocol that transmits and receives data asynchronously, eliminating the need for a shared clock. With support for full-duplex communication, UART enables simultaneous data transmission and reception. Key configuration parameters for the UART include baud rate, data bits, stop bits, parity, and flow control.

This system includes three configurable UART ports— **UART0**, **UART1**, **UART2**, **UART3** and **UART4**—each designed with independent base addresses and control registers. These ports allow seamless integration of multiple serial connections, making the system highly versatile for a variety of communication requirements. Each port supports interrupt-driven operation and features a delay register to fine-tune timing for robust communication in demanding applications.

6.12.1 Instance Details

The UART Ports are identified as UART0, UART1,UART2,UART3 and UART4.

UART Port Address Details

Table 6.12.1: UART Port Register Map

UART Port	Base Address	Interrupt ID
UART0	0x11300	53
UART1	0x11400	54
UART2	0x11500	55
UART3	0x11600	56
UART4	0x11700	57

Note

UART3 and UART4 pins are multiplexed with GPIOs For detailed information on UART pinmux, refer to *Pinmux Register Map*.

6.12.2 Register Map

This section provides the register map with the offsets for each UART Port, including baud rate register (*BAUDREG*) and other important offsets.

Table 6.12.2: UART Register Map

Register Name	Offset (Hex)	Description
BAUD_REG (Baudrate Register)	0x00	Baud rate is a 16-bit Register. Sets the baud rate for UART communication. The value in this register determines the speed of communication (e.g., 9600, 115200, etc.).
TX_REG (TX Register)	0x04	Transmit data register is a 8-bit register. Write data to this register to transmit via UART..
RX_REG (RX Register)	0x08	Receive data register is a 8-bit register. Read data from this register to receive data.
STATUS_REG (Status Register)	0x0c	Status register is 16-bit register. Provides the status of UART operations, including flags like data available, transmit buffer empty, etc.
DELAY_REG (Delay Register)	0x10	Delay register is a 16-bit register. This register can be used for introducing a delay in the UART operation, which can be useful for timing control or ensuring data transmission stability.
CTRL (Control Register)	0x14	Control register is a 16-bit register. Control Register is used to configure various parameters of the UART communication, including character size, parity, and stop bits.
INTR_EN (Interrupt Enable Register)	0x18	Interrupt enable register is a 16-bit register. Allows the enabling or disabling of UART interrupts based on specific conditions (e.g., data availability, transmission completion)

continues on next page

Table 6.12.2 – continued from previous page

Register Name	Offset (Hex)	Description
RX_THRESHOLD (RX Threshold Register)	0x20	Receive threshold is a 8-bit register. Sets the threshold for receiving data, determining how much data should be in the RX buffer before an interrupt is triggered.

Baudrate Register (BAUD_REG)

The **BAUD_REG** is a 16-bit register used to configure the baudrate for UART communication. It controls the speed of data transmission between devices. The value set in this register defines the rate at which data bits are transmitted per second. Common baud rates include values like 9600, 115200, and others, depending on the system's configuration and communication requirements.

$$BAUD\_REG = \left\lceil \frac{CLOCK\_FREQUENCY}{16 \times baudrate} \right\rceil$$

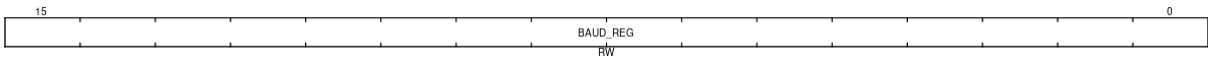


Fig. 6.12.1: Baudrate Register

The exact value entered into this register determines the UART baud rate.

TX Register (TX_REG)

The **TX_REG** is an 32-bit register used for placing data into the UART transmit buffer. When data is written to this register, it is stored in the transmit buffer which can store 16 bytes of data, where it will be serialized and transmitted over the UART interface based on the configured baud rate and communication parameters. This register effectively acts as a staging area for data before it is sent out over the UART line. After the data is placed in the buffer, the UART controller handles the actual transmission process asynchronously.

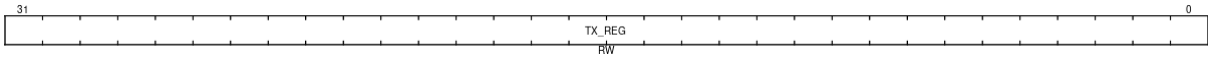


Fig. 6.12.2: TX Register

RX Register (RX_REG)

The **RX_REG** is an 32-bit register used for reading incoming data from the UART receive buffer. Data that is received via UART is first placed into the receive buffer which can store 16 bytes of data, and the **RX_REG** allows software to read this data that is present in the receive buffer.

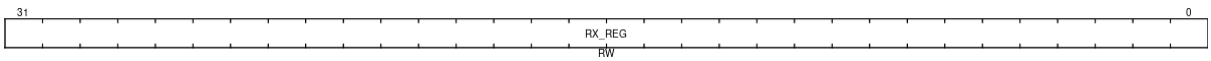


Fig. 6.12.3: RX Register

This register essentially acts as an interface to access the buffered incoming data.

Note

Configure the control register, TX_THRESH and RX_THRES bits in *Control Register*. To write/read 32, 16 or 8 bits directly from the TX/RX register.

Status Register (STATUS_REG)

The **STATUS_REG** holds flags to indicate the current state of the UART interface. These flags inform about transmission and reception status, error conditions, FIFO thresholds, and control delays.

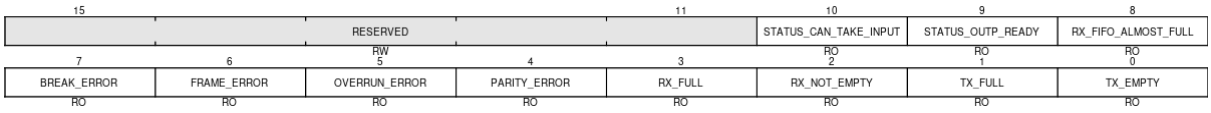


Fig. 6.12.4: Status Register

Table 6.12.3: UART Status Register

Bits	Field Name	Permission	Description
[0:0]	TX_EMPTY	Read-only	This bit is set when the transmitter register is empty and ready to accept new data. It indicates that the UART transmitter is idle and can begin transmitting new data.
[1:1]	TX_FULL	RO	This bit is set when the transmitter register is full and cannot accept more data. The UART is waiting for space to become available in the register before additional data can be sent.
[2:2]	RX_NOT_EMPTY	RO	This bit is set when the receiver register contains data that can be read. It indicates that the UART has received data and it is available for reading.
[3:3]	RX_FULL	RO	This bit is set when the receiver register is full and cannot accept additional data. It indicates that the UART's receive buffer is full and more data cannot be stored until the buffer is read.
[4:4]	PARITY_ERROR	RO	This bit is set if a parity error is detected during the reception of data. A parity error occurs when the received data does not match the expected parity (even or odd). Writing a 0 clears the PARITY_ERROR bit in status register.

continues on next page

Table 6.12.3 – continued from previous page

Bits	Field Name	Permission	Description
[5:5]	OVERRUN_ERROR	RO	This bit is set if there is a buffer overrun error, meaning that data was lost because the receiver buffer was full. New incoming data could not be stored and was discarded. Writing a 0 clears the OVERRUN_ERROR bit in status register.
[6:6]	FRAME_ERROR	RO	This bit is set if the stop bit received is 0. Writing a 0 clears the FRAME_ERROR bit in status register.
[7:7]	BREAK_ERROR	RO	This bit is set if both, the data and the stop bits received are all zeros. Writing a 0 clears the BREAK_ERROR bit in status register.
[8:8]	RX_FIFO_ALMOST_FUL	RO	This bit is set when the receiver FIFO buffer is nearly full. This indicates that data should be read from the buffer before it overflows, preventing data loss. Writing a 0 clears the RX_FIFO_ALMOST_FULL bit in status register.
[9:9]	STATUS_OUTP_READY	RO	This bit is set when the device's output is ready.
[10:10]	STATUS_CAN_TAKE_INPUT	RO	This bit is set when the output buffer is ready to take input. This indicates that the device output is ready to accept new input data.
[15:11]	RESERVED	RW	Reserved for future use.

Delay Register (DELAY_REG)

The **DELAY_REG** is a 16-bit register used to introduce a delay in UART operations. This register can be programmed with a specific value to control timing between UART operations, such as data transmission and reception. By using this register, it is possible to ensure proper timing between data bursts, provide stability during communication, or synchronize the UART interface with other peripherals.

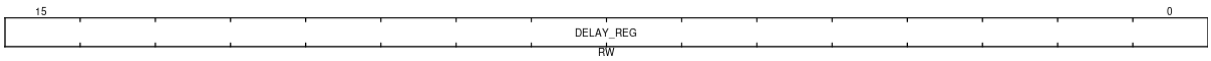


Fig. 6.12.5: Delay Register

The delay set by this register can help manage timing issues or ensure data integrity during high-speed communication.

Control Register (CTRL)

The **CTRL** configures the operating parameters of the UART peripheral. It defines the communication format and behavior of the UART interface, including data length, parity mode, and stop bit configuration.

This register also controls transmitter and receiver enable, transaction bit length, FIFO operation, and internal pull-up resistors on the TX and RX signal lines. Proper configuration of this register ensures reliable UART communication.

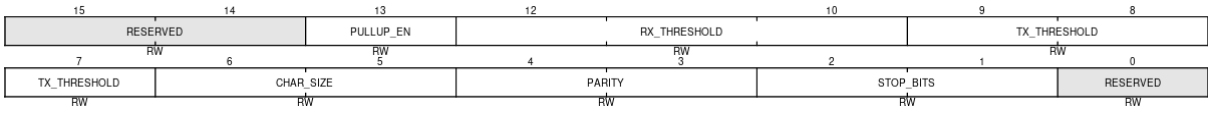


Fig. 6.12.6: Control Register

Table 6.12.4: UART Control Register Flags

Bits	Field Name	Permission	Description
[0:0]	RESERVED	RW	Reserved for future use.

continues on next page

Table 6.12.4 – continued from previous page

Bits	Field Name	Permission	Description
[2:1]	STOP_BITS	RW	Setting these bits configures the number of stop bits used in data transmission: • **00**: 1 Stop bit (Standard configuration for most UART communication) • **01**: 1.5 Stop bits (Used in certain applications requiring extended timing) • **10**: 2 Stop bits (Used for applications requiring more reliable timing)
[4:3]	PARITY	RW	Setting these bits specifies the parity type used for data transmission: • **00**: No parity (Parity checking is disabled, no additional bit for error checking) • **01**: Odd parity (Parity bit ensures the total number of 1s in the byte is odd) • **10**: Even parity (Parity bit ensures the total number of 1s in the byte is even) • **11**: Undefined (This setting is not allowed and should be avoided)

continues on next page

Table 6.12.4 – continued from previous page

Bits	Field Name	Permission	Description
[6:5]	CHAR_SIZE	RW	Setting these bits determines the number of data bits in a character: • **00**: 8 bits (Standard character size for most applications) • **01**: 7 bits (Used for applications where smaller data sizes are required) • **10**: 6 bits (Used for legacy or specialized applications) • **11**: 5 bits (Used for minimal character length communication)
[9:7]	TX_THRESHOLD	RW	Setting these bits determines total number of bits sent in a single frame: • **000**: None • **001**: 8 bits • **010**: 16 bits • **011**: 32 bits
[12:10]	RX_THRESHOLD	RW	Setting these bits determines total number of bits received in a single frame: • **001**: 8 bits • **010**: 16 bits • **011**: 32 bits
[13:13]	PULLUP_EN	RW	Setting this bit enables pull-up resistor on the UART RX line.
[15:14]	RESERVED	RW	Reserved for future use.

Interrupt Enable Register (INTR_EN)

The **INTR_EN** controls the generation of UART interrupts. This register is programmed to enable or disable individual interrupt sources for transmit, receive, and error conditions.

By configuring this register, the UART can operate in interrupt-driven mode, allowing transmit and receive events to be handled by an interrupt service routine instead of polling the status register.

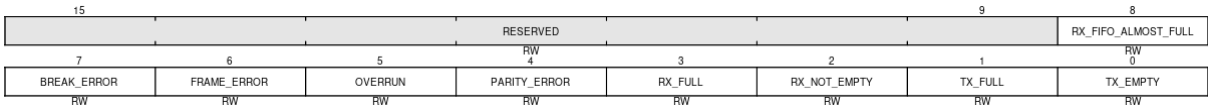


Fig. 6.12.7: Interrupt Enable Register

Bits	Field Name	Permission	Description
[0:0]	TX_DONE	RW	Setting this bit enables the interrupt when the transmission is complete, and the transmit FIFO is empty. The processor will be notified once all data has been transmitted.
[1:1]	TX_NOT_FULL	RW	Setting this bit enables the interrupt when the transmitter register is not full and can accept new data. The processor will be notified when the transmitter is ready to send the next byte.
[2:2]	RX_NOT_EMPTY	RW	Setting this bit enables the interrupt when the receiver register contains data that can be read. The processor will be notified when new data is available for reading.

continues on next page

Table 6.12.5 – continued from previous page

Bits	Field Name	Permission	Description
[3:3]	RX_FULL	RW	Setting this bit enables the interrupt when the receiver register is full and cannot accept additional data. It indicates that the UART's receive buffer is full and more data cannot be stored until the buffer is read.
[4:4]	PARITY_ERROR	RW	Setting this bit enables the interrupt when a parity error is detected during data reception. A parity error occurs if the parity of the received data does not match the expected value (odd/even).
[5:5]	OVERRUN_ERROR	RW	Setting this bit enables the interrupt if a buffer overrun error occurs, meaning that received data was lost because the receiver buffer was full. The processor will be notified of data loss due to the buffer overflow.
[6:6]	FRAME_ERROR	RW	Setting this bit enables the interrupt when a framing error occurs, such as incorrect stop bits or other violations of the expected data frame. The processor will be notified of the error.
[7:7]	BREAK_ERROR	RW	Setting this bit enables the interrupt when a break error occurs in the received data. A break condition occurs when the UART did not detect the expected start bit or when a framing error occurred, triggering a break.

continues on next page

Table 6.12.5 – continued from previous page

Bits	Field Name	Permission	Description
[8:8]	RX_FIFO_ALMOST_FUL	RW	Setting this bit enables the interrupt when the receiver FIFO is nearly full (about 80% full). The processor will be notified to read the data from the FIFO to avoid overflow.
[15:9]	RESERVED	RW	Reserved for future use.

RX Thresold Register (RX_THRESHOLD)

The **RX_THRESHOLD** is an 8-bit register that controls when an interrupt is triggered based on the amount of data in the RX buffer. It helps manage data reception efficiently by reducing interrupt overhead in high-throughput applications.

Fig. 6.12.8: RX Threshold Register

6.12.3 UART Initialization and Operation Workflow

This section describes the UART initialization and data handling workflow for both Transmit and Receive operations. The configuration sequence ensures correct UART setup before enabling data transfer using either polling or interrupt mode.

Common Initialization Steps

These steps are common for both transmit and receive operations.

1. **Set Baud Rate** Configure the UART baud rate register according to the required communication speed.
2. **Configure Control Register** Configure UART control parameters such as: - Configure the total number of bits to be written to or read from the TX/RX register in a single transaction. - Parity: Enable and configure parity generation and checking for transmitted and received data. - Stop bits: Configure the number of stop bits

appended to each transmitted character. - Character size: Configure the number of data bits transmitted from or received into the FIFO per character. - Pull-up: Enable internal pull-up resistors on the TX and RX signal lines when required.

3. **Configure RX Threshold (Optional)** If RX FIFO or threshold-based reception is required, configure the RX threshold value.

After completing the above steps, proceed with either transmit or receive workflow.

UART Transmit Workflow

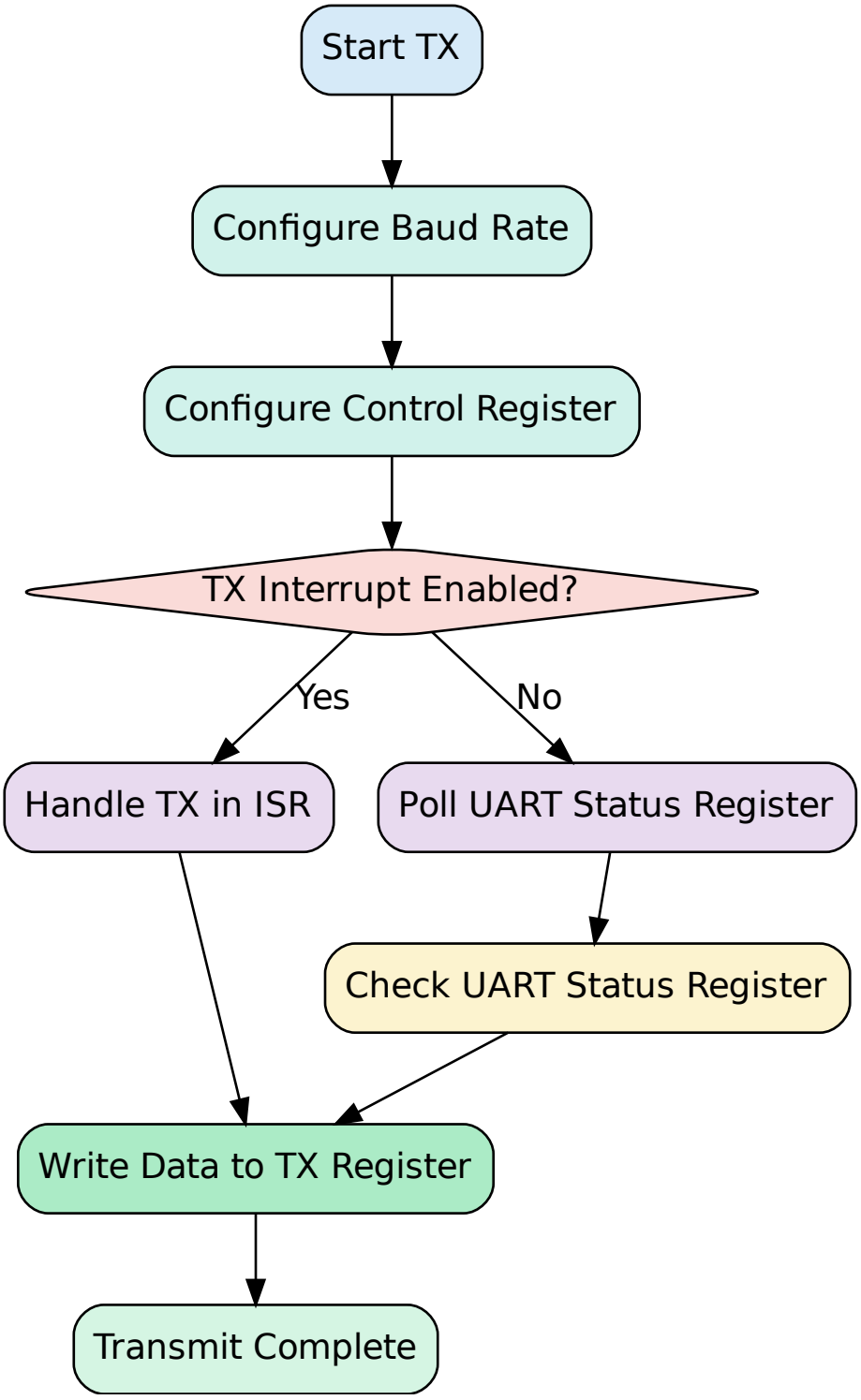
The operating mode is determined by whether the TX interrupt is enabled.

If TX interrupt is enabled, transmission is handled in interrupt mode. If TX interrupt is not enabled, transmission is handled in polling mode.

Transmit Steps

1. **Enable TX Interrupt (Interrupt Mode Only)** If interrupt mode is selected, enable the UART TX interrupt.
2. **Poll Status Register (Polling Mode Only)** If polling mode is selected, poll the TX ready or TX empty status flag.
3. **Write Data to TX Register** Write the transmit data into the UART TX data register.
4. **Transmit Complete** Data transmission is completed once the TX FIFO and shift register are empty.

UART Transmit Flowchart



UART Receive Workflow

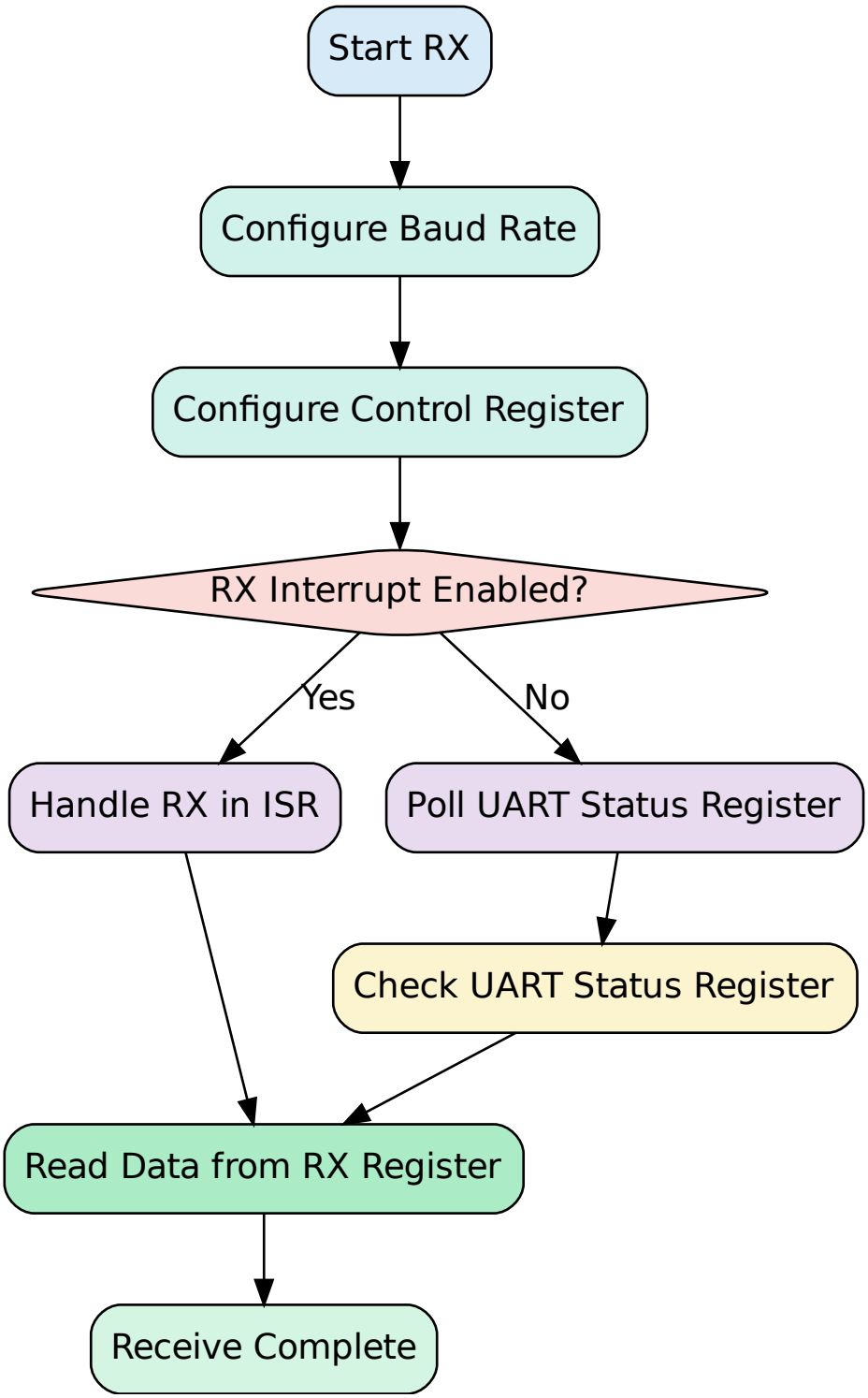
The operating mode is determined by whether the RX interrupt is enabled.

If RX interrupt is enabled, reception is handled in interrupt mode. If RX interrupt is not enabled, reception is handled in polling mode.

Receive Steps

1. **Enable RX Interrupt (Interrupt Mode Only)** If interrupt mode is selected, enable the UART RX interrupt.
2. **Poll Status Register (Polling Mode Only)** If polling mode is selected, poll the RX data available status flag.
3. **Read Data from RX Register** Read received data from the UART RX data register.
4. **Receive Complete** Data reception is completed once the RX FIFO is empty.

UART Receive Flowchart



6.13 Watchdog Timer (WDT)

The Watchdog Timer (WDT) is a hardware safety mechanism that monitors software execution and resets the processor if the watchdog is not serviced within a programmed timeout period. This ensures system recovery to a known and stable state.

The WDT supports the following operating modes:

- **HARD_RESET:** Generates a system-wide reset when the programmed watchdog timeout expires.
- **SOFT_RESET:** Generates a processor (core) reset without resetting the entire system.

6.13.1 Instance Details

The system contains a single Watchdog Timer instance used for watchdog configuration and control.

Table 6.13.1: WDTimer Instance Map

WDTimer Instance	Base Address	Interrupt line
WDT	0x00040500	WDT

Note

The WDTimer interrupt is not connected to the PLIC on S2401. As a result, watchdog-generated interrupts are not captured by the PLIC, even when the WDTimer is operating in interrupt mode.

6.13.2 Register Map and Details

The following table lists the WDTimer registers, their offsets, sizes, and descriptions.

Table 6.13.2: WDTimer Register Map

Register Name	Offset	Size (in bits)	Description
WDT_CYCLES	0x0000	64	Configures the number of clock cycles after which the chip should reset in HARD_RESET mode.
WDT_CTRL	0x0008	16	Enables and configures the WDTimer operating mode.
WDT_ACTIVE	0x0018	32	Updates the internal watchdog counter using `WDT_CYCLES` register.

WDT_CYCLES Register

This 64-bit register configures the number of clock cycles after which the system resets when operating in **HARD_RESET** mode.

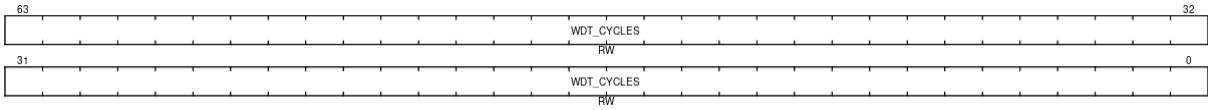


Fig. 6.13.1: WDT_CYCLES Register

WDT_CTRL Register

This 16-bit register enables and configures the operating mode of the Watchdog Timer.

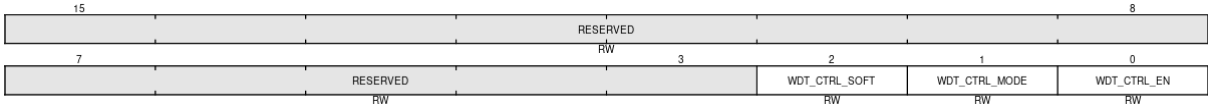


Fig. 6.13.2: WDT_CTRL Register

Table 6.13.3: Register Bit Fields

Bits	Field Name	Permission	Description
[0:0]	WDT_CTRL_EN	RW	Enables the watchdog timer.
[1:1]	WDT_CTRL_MODE	RW	Selects interrupt(0) or hard reset mode(1).
[2:2]	WDT_CTRL_SOFT	RW	Enables soft reset mode.
[15:3]	RESERVED	RW	Reserved for future use.

WDT_ACTIVE Register

Writing `1` updates the internal watchdog counter using `WDT_CYCLES`.

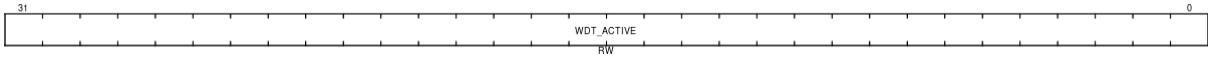


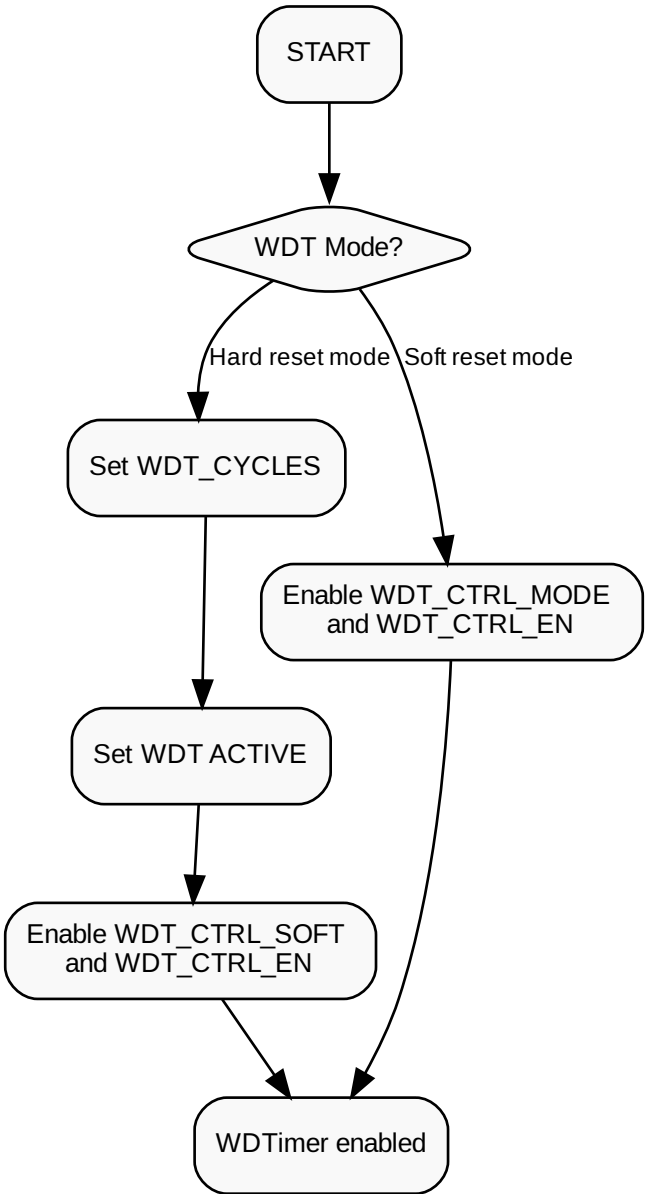
Fig. 6.13.3: WDT_ACTIVE Register

6.13.3 Workflow

6.13.4 WDT Operation

1. Program the watchdog timeout value in the WDT_CYCLES register based on the required timeout period and the system clock frequency.
2. Select the watchdog operating mode as either hardware reset mode or software reset (interrupt) mode.
3. If hardware reset mode is selected, set the HARD_RESET mode in the WDT_CTRL register.
4. If software reset mode is selected, set the SOFT_RESET mode in the WDT_CTRL register.
5. Configure the WDT_CTRL register with the selected reset mode and required control options.
6. Enable the watchdog timer by setting the watchdog enable bit in the control register.

7. Disable the watchdog timer by setting the EN bit in the WDT_CTRL register, when reset is no longer needed.



Chapter 7. Security Accelerators

7.1 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a symmetric-key block cipher used for both the encryption and decryption of sensitive data. As a symmetric algorithm, it utilizes the same secret key to transform plaintext into ciphertext and to revert ciphertext back to its original form.

The **S2401** features a dedicated **AES Hardware Accelerator**, designed to offload cryptographic workloads from the main processor. This hardware implementation supports **AES with 128-, 192-, and 256-bit keys** and provides native support for four standard modes of operation:

- **CBC** (Cipher Block Chaining)
- **CFB** (Cipher Feedback)
- **OFB** (Output Feedback)
- **CTR** (Counter Mode)

The supported key lengths are:

- **128 bits (16 bytes)**
- **192 bits (24 bytes)**
- **256 bits (32 bytes)**

By utilizing these hardware-accelerated modes, the system ensures high-throughput data processing and improved power efficiency.

7.1.1 Instance Details

The **S2401** incorporates a single hardware instance of the AES accelerator.

Table 7.1.1: AES Instance Map

AES Instance	Base Address	Interrupt ID
AES	0x04000000UL	NA

7.1.2 Register Map

This section provides the register map for the AES hardware accelerator. The data registers are 64 bits wide, while the control and status registers are 8 bits wide.

Table 7.1.2: AES Register Map

Register Name	Offset	Length (bits)	Description
INPUT	0x0000	64	Data input register for the plaintext or ciphertext block.
KEY	0x0020	64	Key data register for loading the cryptographic key.
OUTPUT	0x0040	64	Output data register holding the result of the AES operation.
IV	0x0050	64	Initialization vector register for chained modes of operation.
CTRL	0x0060	8	Control register for configuring the operation mode, key length, and direction.
STATUS	0x0061	8	Status register indicating input readiness and output availability.
ZEROIZE	0x0074	8	Write register to initiate secure clearing of all AES accelerator registers.
ZEROIZE_STATUS	0x0078	8	Status register indicating the completion of the zeroize operation.

7.1.3 Register Details

Input Register (INPUT)

The 64-bit **INPUT** register is the entry point for loading the plaintext or ciphertext block into the AES hardware accelerator. The AES engine processes data in 128-bit blocks, requiring **2 consecutive 64-bit writes** to this register address.

Data must be provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

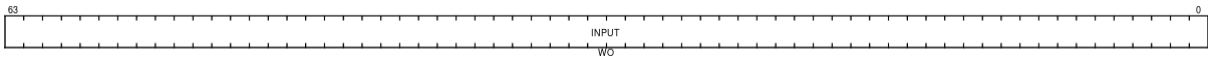


Fig. 7.1.1: INPUT

Key Data Register (KEY)

The 64-bit **KEY** register is used to load the cryptographic key into the AES hardware accelerator. The number of writes required depends on the key length configured in the `CTRL_KEYLEN` field of the CTRL register.

For key lengths shorter than 256 bits, the most significant register slots must be zero-padded before writing the actual key material:

- **128-bit key** — 2 zero-padded writes followed by 2 key writes (4 writes total)
- **192-bit key** — 1 zero-padded write followed by 3 key writes (4 writes total)
- **256-bit key** — 4 key writes (no zero-padding required)

Data must be provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

Fig. 7.1.2: KEY

Initialization Vector Register (IV)

The 64-bit **IV** register is used to load the 128-bit Initialization Vector required for chained modes of operation (CBC, CFB, OFB, and CTR). The full 128-bit IV is loaded by performing **2 consecutive 64-bit writes** to this register address. This register must be loaded before initiating the AES operation.

Data must be provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

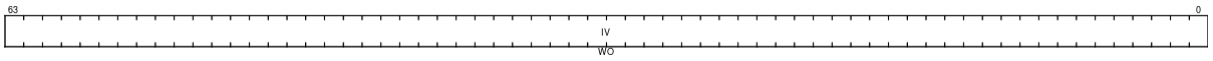


Fig. 7.1.3: IV

Output Data Register (OUTPUT)

The 64-bit **OUTPUT** register holds the 128-bit result of the AES encryption or decryption operation. The result is available for reading once the **STATUS_OUTP_READY** bit in the STATUS register is set. The full 128-bit result is retrieved by performing **2 consecutive 64-bit reads** from this register address.

Data is provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

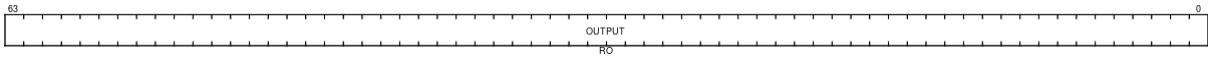


Fig. 7.1.4: OUTPUT

Control Register (CTRL)

The 8-bit **CTRL** register is used to configure the operation mode, key length, and encryption direction of the AES hardware accelerator. This register must be written before loading the key and IV registers.

Fig. 7.1.5: CTRL

Table 7.1.3: CTRL

Bits	Field Name	Permission	Description
[0:0]	CTRL_ENCDEC	RW	Selects the direction of the operation. Set to **0** for encryption and **1** for decryption.

continues on next page

Table 7.1.3 – continued from previous page

Bits	Field Name	Permission	Description
[2:1]	CTRL_KEYLEN	RW	Selects the key length. Set to **0** for 128-bit, **1** for 192-bit, and **2** for 256-bit.
[5:3]	CTRL_MODE	RW	Selects the block cipher mode of operation. Set to **1** for CBC, **2** for CFB, **3** for OFB, and **4** for CTR.
[6:6]	CTRL_END	RW	Set to **1** to indicate that the final block of the message is currently being processed.
[7:7]	RESERVED	RW	Reserved for future use.

Status Register (STATUS)

The 8-bit **STATUS** register is a read-only register used to monitor the operational state of the AES hardware accelerator.

Fig. 7.1.6: STATUS

Table 7.1.4: STATUS

Bits	Field Name	Permission	Description
[0:0]	STATUS_CAN_TAKI	RO	When set to **1**, indicates that the accelerator is ready to accept the next 64-bit data block.
[1:1]	STATUS_OUTP_REA	RO	When set to **1**, indicates that the AES operation is complete and the result is available to be read from the OUTPUT register.
[7:2]	RESERVED	RW	Reserved for future use.

Zeroize Register (ZEROIZE)

The 8-bit **ZEROIZE** register is used to securely clear all sensitive data held within the AES hardware accelerator registers. Writing **1** to this register initiates the zeroize operation. Once triggered, the completion of the operation must be confirmed by polling the **ZEROIZE_STATUS** register before initiating any subsequent AES operation.

Fig. 7.1.7: ZEROIZE

Table 7.1.5: ZEROIZE

Bits	Field Name	Permission	Description
[0:0]	ZEROIZE	WO	Write **1** to initiate the zeroize operation. All sensitive data across the AES hardware accelerator registers will be securely cleared.
[7:1]	RESERVED	RW	Reserved for future use.

Zeroize Status Register (ZEROIZE_STATUS)

The 8-bit **ZEROIZE_STATUS** register indicates the current status of the zeroize operation. This register must be polled after writing to the ZEROIZE register to confirm that the operation has completed before proceeding with any subsequent AES operation.

Fig. 7.1.8: ZEROIZE_STATUS

Table 7.1.6: ZEROIZE_STATUS

Bits	Field Name	Permission	Description
[0:0]	ZEROIZE_BUSY	RO	Set to **1** while the zeroize operation is in progress. Cleared to **0** once the operation is complete and the hardware is ready for the next operation.
[7:1]	RESERVED	RW	Reserved for future use.

7.1.4 Workflow

Operation Sequence

The AES hardware execution involves zeroization, configuration, key and IV loading, block-by-block data processing, and result retrieval.

1. Zeroization

Before the first operation, write **1** to the ZEROIZE register to securely clear all internal registers. Poll `ZEROIZE_STATUS[0]` (ZEROIZE_BUSY) until it is cleared to **0** to confirm completion.

2. Configure CTRL Register

Write to the CTRL register to set the encryption direction, key length, and mode of operation. This register must be configured before loading the key and IV.

3. Load KEY Register

Load the cryptographic key by performing **4 consecutive 64-bit writes** to the KEY register address. For key lengths shorter than 256 bits, the most significant register slots must be zero-padded before writing the key material.

4. Load IV Register

Load the 128-bit Initialization Vector by performing **2 consecutive 64-bit writes** to the IV register address.

5. Load INPUT Register

Pack the 128-bit input block into 64-bit words in **big-endian byte order** and perform **2 consecutive 64-bit writes** to the INPUT register address.

6. Poll STATUS Register

Poll `STATUS[1]` (STATUS_OUTP_READY) until it is set to **1**, indicating that the AES operation on the current block is complete.

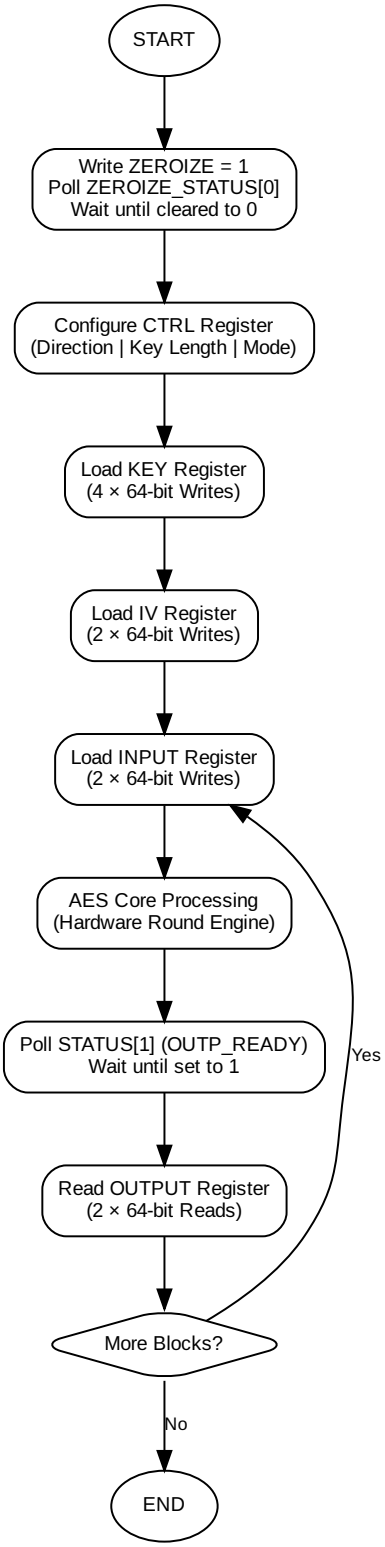
7. Read OUTPUT Register

Perform **2 consecutive 64-bit reads** from the OUTPUT register address and unpack each 64-bit word from big-endian byte order to assemble the 128-bit result.

8. Repeat for Remaining Blocks

If additional blocks remain, repeat steps 5 through 7 for each subsequent block. The hardware automatically manages the chaining state between blocks.

AES Operation Flowchart



7.2 One-Time Programmable Memory(OTP)

The One-Time Programmable (OTP) unit is a High-Density Non-Volatile Memory (NVM) IP block. Based on the XPM (eXtended Programming Matrix) architecture, it provides a secure and reliable permanent storage solution for critical system data. It is designed to permanently store configuration data, security keys, calibration values, and other critical information that must be retained across power cycles. Once programmed, the contents of the OTP memory cannot be modified or erased, ensuring data integrity and protection against unintended changes.

7.2.1 Instance Details

The table below lists the OTP instances and their respective base addresses.

Table 7.2.1: OTP Instance Details

OTP Instance	Base Address
OTP_BASE_ADDRESS	0x00033000

7.2.2 Register Map and Details

The following table lists the registers of the OTP controller along with their offsets and functional descriptions.

Table 7.2.2: OTP Register Map

Register Name	Offset	Length (In Bits)	Description
CTRL (Control Register)	0x00	32	Configures and initiates OTP initialization, read, and write operations.
STATUS (Status Register)	0x08	32	Reports the current operational state and error conditions of the OTP controller.

continues on next page

Table 7.2.2 – continued from previous page

Register Name	Offset	Length (In Bits)	Description
ADDRESS (Address Register)	0x10	32	Specifies the OTP memory address for subsequent read or write operations.
DATA_READ (Read Data Register)	0x18	32	Contains the data read from the selected OTP memory address.
DATA_WRITE (Write Data Register)	0x20	32	Holds the data value to be programmed into the selected OTP memory address.

Control Register (CTRL)

The CTRL register is a 32-bit register that configures and controls OTP operations. It is used to initialize the OTP memory, select the operation type (read or write), and start the requested OTP operation.

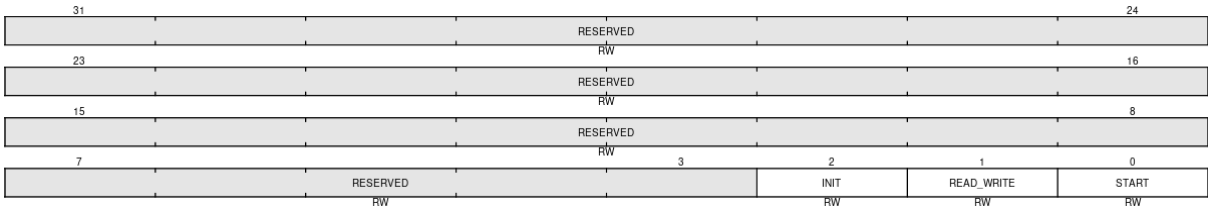


Fig. 7.2.1: CTRL

Table 7.2.3: Control Register Fields

Bits	Field Name	Permission	Description
[0:0]	START	RW	Initiates the OTP operation when set to 1.
[1:1]	READ_WRITE	RW	Selects the operation type: 0 for Write operation and 1 for Read operation

continues on next page

Table 7.2.3 – continued from previous page

Bits	Field Name	Permission	Description
[2:2]	INIT	RW	Initializes the OTP memory before any programming operation.
[31:3]	RESERVED	RW	Reserved

Status Register (STATUS)

The STATUS register is a 32-bit register that provides information about the OTP controller, including operation completion, programming success, and error conditions.

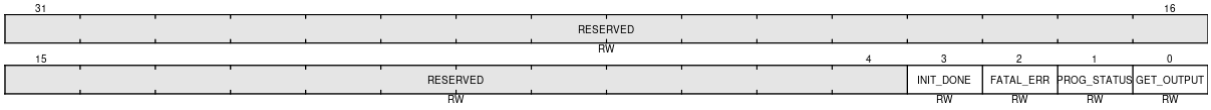


Fig. 7.2.2: STATUS

Table 7.2.4: Status Register Fields

Bits	Field Name	Permission	Description
[0:0]	GET_OUTPUT	RW	Indicates that valid read data is available from the OTP memory.
[1:1]	PROG_STATUS	RW	Indicates the programming status of the OTP memory.
[2:2]	FATAL_ERR	RW	Indicates a fatal error occurred during an OTP programming operation.
[3:3]	INIT_DONE	RW	Indicates completion of OTP memory initialization.
[31:4]	RESERVED	RW	Reserved

Address Register (ADDRESS)

The ADDRESS register is a 32-bit register that specifies the OTP memory address targeted for subsequent read or write operations.

Fig. 7.2.3: ADDRESS

Table 7.2.5: Address Register Fields

Bits	Field Name	Permission	Description
[31:0]	ADDR	RW	Specifies the OTP memory address for the read or write operation.

Read Data Register (DATA_READ)

The DATA_READ register is a 32-bit register that contains data retrieved from the OTP memory following a successful read operation.

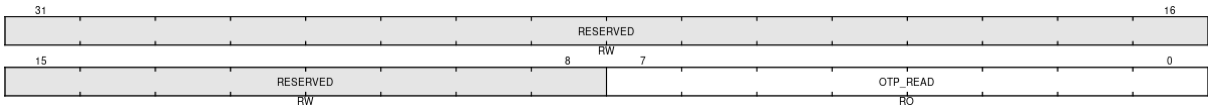


Fig. 7.2.4: DATA_READ

Table 7.2.6: Read Register Fields

Bits	Field Name	Permission	Description
[7:0]	OTP_READ	RO	Contains the 8-bit data value read from the OTP memory.
[31:8]	RESERVED	RW	Reserved

Write Data Register (DATA_WRITE)

The DATA_WRITE register is a 32-bit register that holds the data value to be programmed into the OTP memory during a write operation.

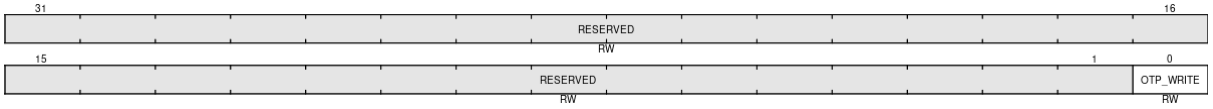


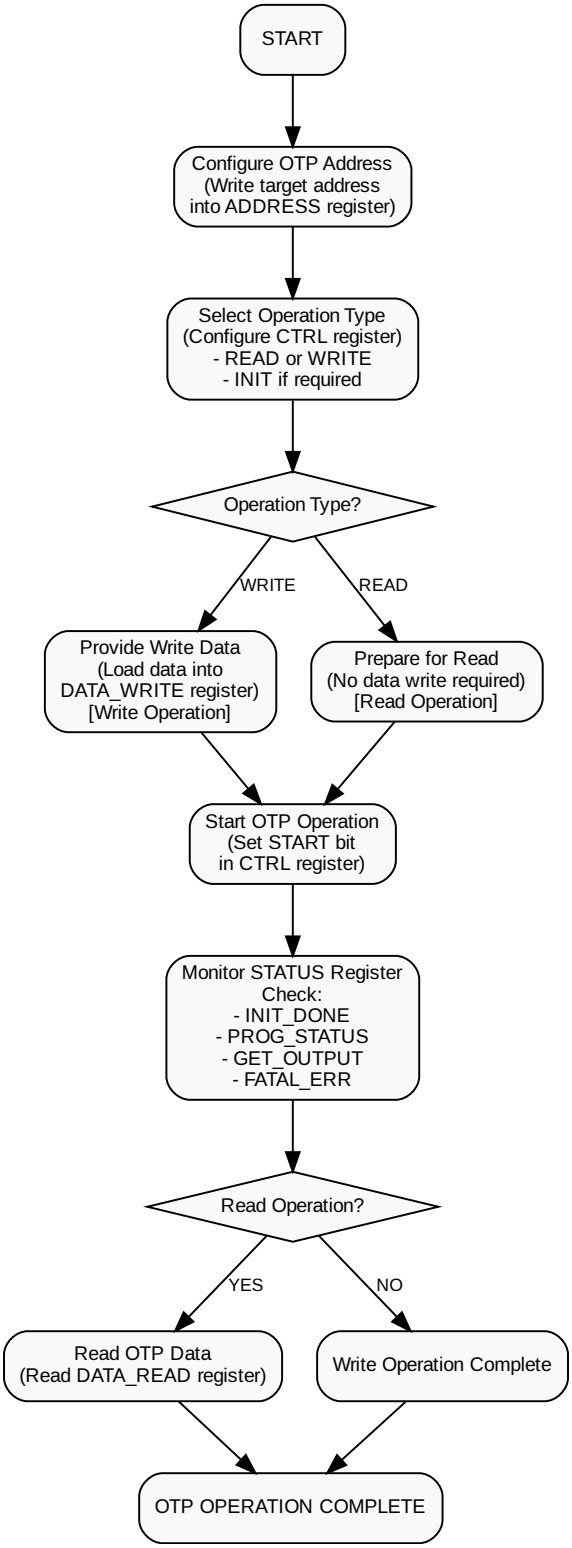
Fig. 7.2.5: DATA_WRITE

Table 7.2.7: Control Register Fields

Bits	Field Name	Permission	Description
[0:0]	OTP_WRITE	RW	Data value to be programmed into the OTP memory.
[31:1]	RESERVED	RW	Reserved

7.2.3 Workflow

1. **Configure OTP Address:** Program the target OTP memory location by writing the required address to the ADDRESS register.
2. **Select Operation Type:** Configure the CTRL register to select the desired operation:
   - Set READ_WRITE to choose between read or write
   - Set INIT if OTP initialization is required before programming
3. **Provide Write Data:** For write operations, load the data to be programmed into the DATA_WRITE register.
4. **Start OTP Operation:** Set the START bit in the CTRL register to initiate the selected OTP operation.
5. **Monitor Operation Status:** Poll the STATUS register to determine:
   - Completion of initialization (INIT_DONE)
   - Availability of read data (GET_OUTPUT)
   - Programming success (PROG_STATUS)
   - Error conditions (FATAL_ERR)
6. **Read OTP Data:** Once GET_OUTPUT is asserted, read the retrieved data from the DATA_READ register.


START
Configure OTP Address
(Write target address
into ADDRESS register)
Select Operation Type
(Configure CTRL register)
- READ or WRITE
- INIT if required
Operation Type?
WRITE
READ
Provide Write Data
(Load data into
DATA_WRITE register)
[Write Operation]
Prepare for Read
(No data write required)
[Read Operation]
Start OTP Operation
(Set START bit
in CTRL register)
Monitor STATUS Register
Check:
- INIT_DONE
- PROG_STATUS
- GET_OUTPUT
- FATAL_ERR
Read Operation?
YES
NO
Read OTP Data
(Read DATA_READ register)
Write Operation Complete
OTP OPERATION COMPLETE

7.3 Rivest-Shamir-Adleman 2048 bits (RSA)

The RSA algorithm is a cornerstone of asymmetric cryptography, widely used for secure data transmission, digital signatures, and key exchange. It relies on the mathematical difficulty of factoring the product of two large prime numbers. A 2048-bit key length provides a high security margin, making it suitable for modern security standards.

The **S2401** features a specialized **RSA-2048 Hardware Accelerator**. This engine is designed to handle the computationally intensive modular exponentiation (a^b mod n) required by RSA. By performing these operations in dedicated hardware, the system avoids the heavy performance penalties associated with software-based BigNum libraries.

7.3.1 Instance Details

The **S2401** incorporates a single hardware instance of the RSA-2048 accelerator.

Table 7.3.1: RSA Instance Map

RSA Instance	Base Address	Interrupt ID
RSA	0x05000000UL	NA

7.3.2 Register Map

This section provides the register map for the RSA hardware accelerator. All data registers are 64 bits wide and must be accessed sequentially with 32 iterations to process the full 2048-bit (256-byte) data values.

Table 7.3.2: RSA Register Map

Register Name	Offset	Length (bits)	Description
INPUT	0x0000	64	Data input register for the base value of the modular exponentiation operation.
EXP	0x0020	64	Exponent register holding the public or private key exponent.
MOD	0x0040	64	Modulus register holding the 2048-bit modulus (n).
R2MODN	0x0060	64	Montgomery constant register holding the precomputed value R^2 mod N.
OUTPUT	0x0080	64	Output register holding the 2048-bit result of the modular exponentiation.
STATUS	0x00C0	8	Read-only status register indicating output readiness and accelerator state.
ZEROIZE	0x00C4	8	Write register to initiate secure clearing of all RSA accelerator registers.
ZEROIZE_STATUS	0x00C8	8	Read-only status register indicating the completion of the zeroize operation.

7.3.3 Register Details

Input Register (INPUT)

The 64-bit **INPUT** register holds the 2048-bit base value for the modular exponentiation operation (a^b mod n). Depending on the RSA operation being performed, this value represents the plaintext, ciphertext, message hash, or signature.

To load the full 2048-bit value, **32 consecutive 64-bit writes** must be performed to this register address. Data must be provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

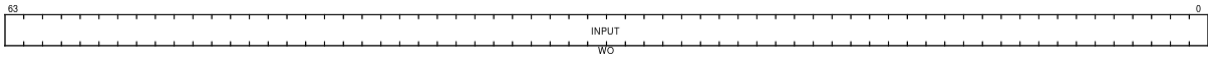


Fig. 7.3.1: INPUT

Exponent Register (EXP)

The 64-bit **EXP** register holds the 2048-bit exponent value for the modular exponentiation operation (a^b mod n). Depending on the RSA operation being performed, this value represents the public exponent (e) during encryption or signature verification, or the private exponent (d) during decryption or signing.

To load the full 2048-bit value, **32 consecutive 64-bit writes** must be performed to this register address. Data must be provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

Fig. 7.3.2: EXP

Modulus Register (MOD)

The 64-bit **MOD** register holds the 2048-bit modulus value (n) for the modular exponentiation operation (a^b mod n). The modulus is a product of two large prime numbers and is common to both the public and private keys in an RSA key pair.

To load the full 2048-bit value, **32 consecutive 64-bit writes** must be performed to this register address. Data must be provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

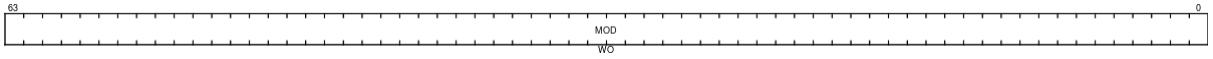


Fig. 7.3.3: MOD

R^2 mod N Register (R2MODN)

The 64-bit **R2MODN** register holds the 2048-bit precomputed Montgomery constant R^2 mod N, where R = 2^2048. This value is derived from the modulus (N) and is required by the hardware to perform Montgomery modular multiplication internally.

This value must be computed from the modulus before initiating the RSA operation and loaded into this register prior to execution.

To load the full 2048-bit value, **32 consecutive 64-bit writes** must be performed to this register address. Data must be provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

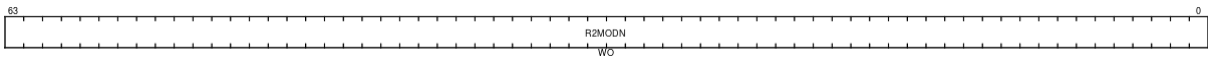


Fig. 7.3.4: R2MODN

Output Register (OUTPUT)

The 64-bit **OUTPUT** register holds the 2048-bit result of the modular exponentiation operation (a^b mod n). Depending on the RSA operation performed, this value represents the encrypted ciphertext, decrypted plaintext, generated signature, or verified message.

The result is available for reading once the **OUTP_READY** bit in the STATUS register is set. To read the full 2048-bit result, **32 consecutive 64-bit reads** must be performed from this register address. Data is provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

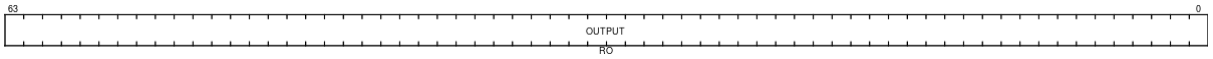


Fig. 7.3.5: OUTPUT

Status Register (STATUS)

The 8-bit **STATUS** register is a read-only register used to monitor the operational state of the RSA hardware accelerator.

Fig. 7.3.6: STATUS

Table 7.3.3: STATUS

Bits	Field Name	Permission	Description
[0:0]	OUTP_READY	RO	When set to 1, indicates that the modular exponentiation is complete and the result is available to be read from the OUTPUT register.
[1:1]	STATUS_READY	RO	When set to 1, indicates that the RSA hardware accelerator is idle and ready to accept a new set of inputs.
[7:2]	RESERVED	RW	Reserved for future use.

Zeroize Register (ZEROIZE)

The 8-bit **ZEROIZE** register is used to securely clear all sensitive data held within the RSA hardware accelerator registers. Writing **1** to this register initiates the zeroize operation. Once triggered, the completion of the operation must be confirmed by polling the **ZEROIZE_STATUS** register before initiating any subsequent RSA operation.

Fig. 7.3.7: ZEROIZE

Table 7.3.4: ZEROIZE

Bits	Field Name	Permission	Description
[0:0]	ZEROIZE	WO	Write **1** to initiate the zeroize operation. All sensitive data across the RSA hardware accelerator registers will be securely cleared.
[7:1]	RESERVED	RO	Reserved for future use.

Zeroize Status Register (ZEROIZE_STATUS)

The 8-bit **ZEROIZE_STATUS** register indicates the current status of the zeroize operation. This register must be polled after writing to the ZEROIZE register to confirm that the operation has completed before proceeding with any subsequent RSA operation.

Fig. 7.3.8: ZEROIZE_STATUS

Table 7.3.5: ZEROIZE_STATUS

Bits	Field Name	Permission	Description
[0:0]	ZEROIZE_BUSY	RO	Set to **1** while the zeroize operation is in progress. Cleared to **0** once the operation is complete and the hardware is ready for the next operation.
[7:1]	RESERVED	RO	Reserved for future use.

7.3.4 Workflow

Operation Sequence

The RSA hardware execution involves precomputation, multi-word data loading, synchronous result retrieval, and post-operation zeroization.

1. Software Precomputation

Before loading data, the Montgomery constant R^2 mod N must be computed from the modulus. This is a one-time computation per unique modulus.

2. Data Loading (Word-by-Word)

For each of the four input registers (INPUT, EXP, MOD, and R2MODN):

- Pack 8 consecutive bytes into a 64-bit word in **big-endian byte order**, with the most significant byte at bits [63:56].

- Write 32 words to the respective register address.

3. Execution and Polling

The hardware automatically begins processing once all four registers are fully loaded. Poll `STATUS[0]` (OUTP_READY) until it is set to **1** to detect completion.

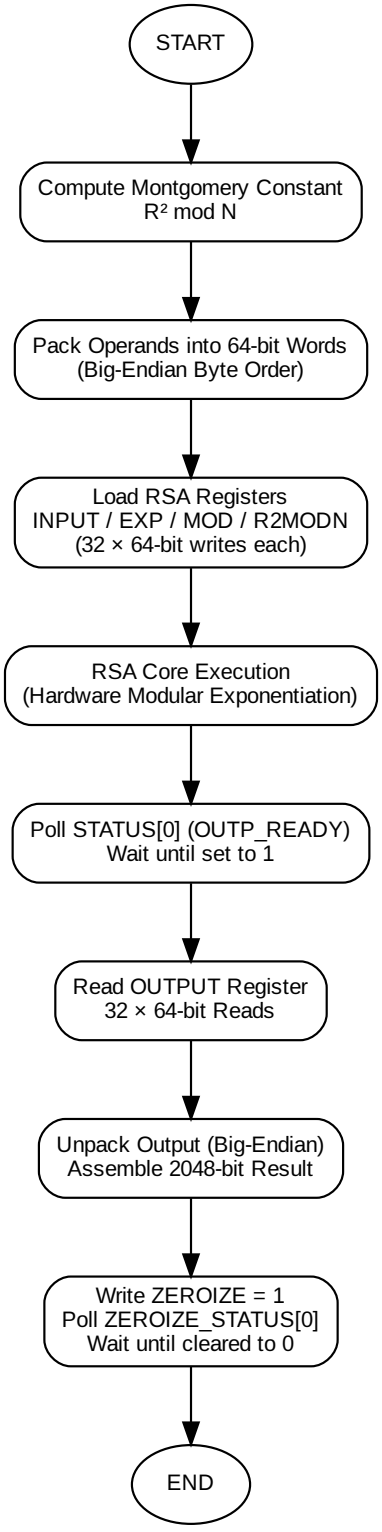
4. Result Extraction

Read the `OUTPUT` register **32 consecutive times**. Unpack each 64-bit word from big-endian byte order to assemble the final 2048-bit result.

5. Zeroization

Write **1** to the ZEROIZE register to securely erase all sensitive data from the RSA hardware accelerator registers. Poll `ZEROIZE_STATUS[0]` (ZEROIZE_BUSY) until it is cleared to **0** to confirm that zeroization is complete.

RSA Operation Flowchart



7.4 Secure Hashing Algorithm 256 bits (SHA256)

The Secure Hash Algorithm 256 (SHA-256) is a cryptographic hash function that produces a unique, fixed-size 256-bit (32-byte) message digest from variable-length input data. It serves as a foundational primitive for data integrity verification, digital signature generation, and secure boot sequences.

The **S2401** integrates a dedicated **SHA-256 Hardware Accelerator** designed to execute SHA-256 operations with minimal CPU overhead. By offloading the computationally intensive compression functions and message scheduling to dedicated hardware, the system achieves superior throughput and significantly lower power consumption compared to software-based implementations.

This accelerator is fully compliant with the **FIPS PUB 180-4 Secure Hash Standard**, ensuring a consistent 256-bit output for any input length.

7.4.1 Instance Details

The **S2401** incorporates a single hardware instance of the SHA-256 accelerator.

Table 7.4.1: SHA256 Instance Map

SHA256 Instance	Base Address	Interrupt ID
SHA256	0x03000000UL	NA

7.4.2 Register Map

This section provides the register map for the SHA-256 hardware accelerator. The data registers are 64 bits wide, while the control and status registers are 8 bits wide.

Table 7.4.2: SHA256 Register Map

Register Name	Offset	Length (bits)	Description
INPUT	0x0000	64	Data input register for the message to be hashed.
OUTPUT	0x0080	64	Output register holding the 256-bit message digest.
CTRL	0x00C0	8	Control register for configuring single-block and multi-block hashing operations.
STATUS	0x00C1	8	Status register indicating input readiness and output availability.
ZEROIZE	0x00C4	8	Write register to initiate secure clearing of all SHA-256 accelerator registers.
ZEROIZE_STATUS	0x00C8	8	Status register indicating the completion of the zeroize operation.

7.4.3 Register Details

Input Register (INPUT)

The 64-bit **INPUT** register is the entry point for loading the message data into the SHA-256 hardware accelerator. The SHA-256 engine processes data in 512-bit blocks, requiring **8 consecutive 64-bit writes** to this register address.

Data must be provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

The SHA-256 engine **automatically begins processing** once the final 64-bit word of the 512-bit block is written to this register. No explicit start trigger is required.

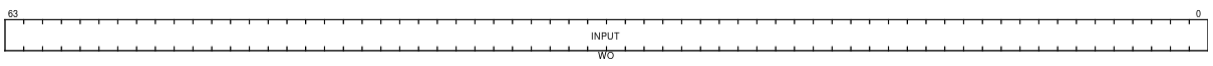


Fig. 7.4.1: INPUT

Output Register (OUTPUT)

The 64-bit **OUTPUT** register holds the 256-bit message digest produced by the SHA-256 hardware accelerator. The result is available for reading once the **STATUS_OUT_READY** bit in the STATUS register is set. The full 256-bit digest is retrieved by performing **4 consecutive 64-bit reads** from this register address.

Data is provided in **big-endian byte order**, where each 64-bit word is formed from 8 consecutive bytes with the most significant byte at bits [63:56].

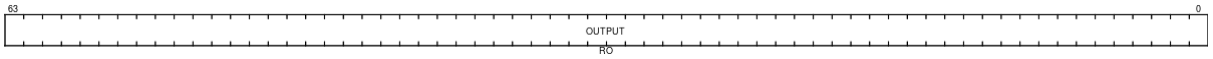


Fig. 7.4.2: OUTPUT

Control Register (CTRL)

The 8-bit **CTRL** register is used to configure the hashing session, specifically for handling multi-block message processing.

Fig. 7.4.3: CTRL

Table 7.4.3: CTRL

Bits	Field Name	Permission	Description
[0:0]	CTRL_CONT_PI	RW	Determines the starting hash value. Set to **0** to start a new hash operation with the initial SHA-256 pre-hash values. Set to **1** to continue hashing from the intermediate state of the previous block, used for multi-block message processing.
[7:1]	RESERVED	RW	Reserved for future use.

Status Register (STATUS)

The 8-bit **STATUS** register is a read-only register used to monitor the operational state of the SHA-256 hardware accelerator.

Fig. 7.4.4: STATUS

Table 7.4.4: STATUS

Bits	Field Name	Permission	Description
[0:0]	STATUS_READY	RO	When set to **1**, indicates that the SHA-256 engine is ready to accept the next 64-bit input block.
[1:1]	STATUS_OUT_REAI	RO	When set to **1**, indicates that the hashing operation is complete and the message digest is available to be read from the OUTPUT register.
[7:2]	RESERVED	RW	Reserved for future use.

Zeroize Register (ZEROIZE)

The 8-bit **ZEROIZE** register is used to securely clear all sensitive data held within the SHA-256 hardware accelerator registers. Writing **1** to this register initiates the zeroize operation. Once triggered, the completion of the operation must be confirmed by polling the **ZEROIZE_STATUS** register before initiating any subsequent SHA-256 operation.

Fig. 7.4.5: ZEROIZE

Table 7.4.5: ZEROIZE

Bits	Field Name	Permission	Description
[0:0]	ZEROIZE	WO	Write **1** to initiate the zeroize operation. All sensitive data across the SHA-256 hardware accelerator registers will be securely cleared.
[7:1]	RESERVED	RW	Reserved for future use.

Zeroize Status Register (ZEROIZE_STATUS)

The 8-bit **ZEROIZE_STATUS** register indicates the current status of the zeroize operation. This register must be polled after writing to the ZEROIZE register to confirm that the operation has completed before proceeding with any subsequent SHA-256 operation.

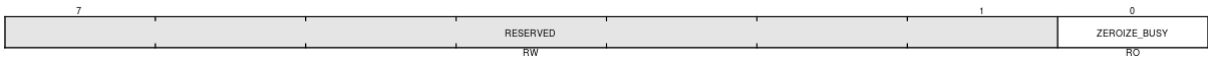


Fig. 7.4.6: ZEROIZE_STATUS

Table 7.4.6: ZEROIZE_STATUS

Bits	Field Name	Permission	Description
[0:0]	ZEROIZE_BUSY	RO	Set to **1** while the zeroize operation is in progress. Cleared to **0** once the operation is complete and the hardware is ready for the next operation.
[7:1]	RESERVED	RW	Reserved for future use.

7.4.4 Workflow

Operation Sequence

The SHA-256 hardware execution involves zeroization, software preprocessing, block-by-block data loading, hardware auto-execution, and result retrieval. Two modes of operation are supported: **Single Run** for processing a complete message in one call, and **Multi Run** for processing large messages incrementally across multiple calls.

1. Zeroization

Before the first operation, write **1** to the ZEROIZE register to securely clear all internal registers. Poll `ZEROIZE_STATUS[0]` (ZEROIZE_BUSY) until it is cleared to **0** to confirm completion.

2. Software Preprocessing

Before loading data into the hardware registers, the following must be computed in software:

- **Total block count** — determines the number of 512-bit blocks required to process the full message.
- **Padding** — generates the mandatory append bits comprising the 0x80 marker byte, zero fill, and the 64-bit big-endian message length field as required by the SHA-256 specification.

3. Configure CTRL Register

For multi-block messages, set `CTRL_CONT_PREHASH` to **1** after the first block is processed to continue hashing from the intermediate state of the previous block. For the first block or a single-block message, this field must be set to **0**.

4. Load INPUT Register

Pack the 512-bit input block into 64-bit words in **big-endian byte order** and perform **8 consecutive 64-bit writes** to the INPUT register address. The SHA-256 engine automatically begins processing once the final 64-bit word is written.

5. Poll STATUS Register

Poll `STATUS[1]` (STATUS_OUT_READY) until it is set to **1**, indicating that the compression of the current block is complete.

6. Handle Padding Overflow

If the padding causes the final block to exceed 512 bits, an additional block containing the remaining padding bits must be loaded and processed by repeating steps 4 and 5.

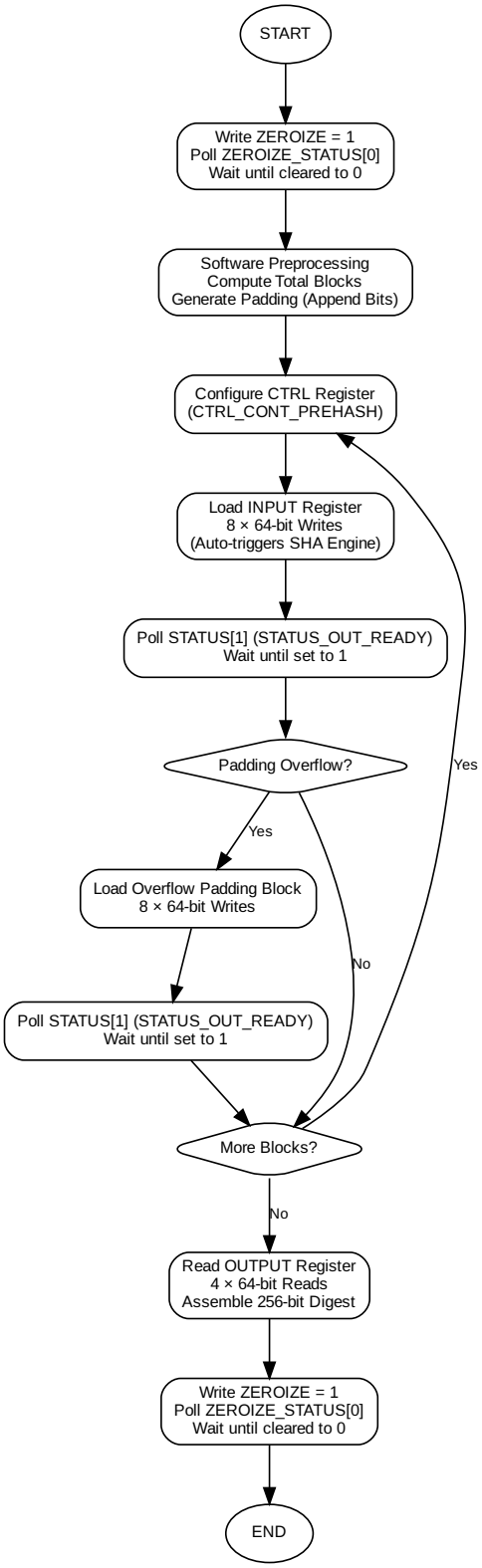
7. Read OUTPUT Register

Once all blocks are processed, perform **4 consecutive 64-bit reads** from the OUTPUT register address and unpack each 64-bit word from big-endian byte order to assemble the final 256-bit message digest.

8. Zeroization

After the output has been read, write **1** to the ZEROIZE register to securely erase all sensitive data from the SHA-256 hardware accelerator registers. Poll `ZEROIZE_STATUS[0]` (ZEROIZE_BUSY) until it is cleared to **0** to confirm that zeroization is complete.

SHA-256 Operation Flowchart



Chapter 8. RISC-V Control and Status Registers

8.1 Control and Status Register(CSR)

8.1.1 CSR Field Specification

The following definitions and abbreviations are used in specifying the behavior of fields within the CSRs.

Reserved Writes Preserve Values, Reads Ignore Values (WPRI)

Some whole read/write fields are reserved for future use. Software should ignore the values read from these fields, and should preserve the values held in these fields when writing values to other fields of the same register. For forward compatibility, implementations that do not furnish these fields must make them read-only zero. These fields are labeled WPRI in the register descriptions.

Write/Read Only Legal Values (WLRL)

Some read/write CSR fields specify behavior for only a subset of possible bit encodings, with other bit encodings reserved. Software should not write anything other than legal values to such a field, and should not assume a read will return a legal value unless the last write was of a legal value, or the register has not been written since another operation (e.g., reset) set the register to a legal value. These fields are labeled WLRL in the register descriptions.

Implementations are permitted but not required to raise an illegal instruction exception if an instruction attempts to write a non-supported value to a WLRL field. Implementations can return arbitrary bit patterns on the read of a WLRL field when the last write was of an illegal value, but the value returned should deterministically depend on the illegal written value and the value of the field prior to the write.

Write Any Values, Reads Legal Values (WARL)

Some read/write CSR fields are only defined for a subset of bit encodings, but allow any value to be written while guaranteeing to return a legal value whenever read. Assuming that writing the CSR has no other side effects, the range of supported values can be determined by attempting to write a desired setting then reading to see if the value was retained. These fields are labeled WARL in the register descriptions.

mplementations will not raise an exception on writes of unsupported values to a WARL field. Implementations can return any legal value on the read of a WARL field when the last write was of an illegal value, but the legal value returned should deterministically depend on the illegal written value and the architectural state of the hart.

8.2 Machine-Level ISA, Version 1.12

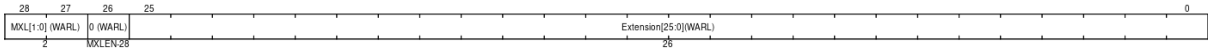
This chapter describes the machine-level operations available in machine-mode (M-mode), which is the highest privilege mode in a RISC-V system. M-mode is used for low-level access to a hardware platform and is the first mode entered at reset. M-mode can also be used to implement features that are too difficult or expensive to implement in hardware directly. The RISC-V machine-level ISA contains a common core that is extended depending on which other privilege levels are supported and other details of the hardware implementation.

8.2.1 Machine-Level CSRs

In addition to the machine-level CSRs described in this section, M-mode code can access all CSRs at lower privilege levels.

Machine ISA Register misa

The misa CSR is a **WARL** read-write register reporting the ISA supported by the hart. This register must be readable in any implementation, but a value of zero can be returned to indicate the misa register has not been implemented, requiring that CPU capabilities be determined through a separate non-standard mechanism.



The MXL (Machine XLEN) field encodes the native base integer ISA width. The MXL

field may be writable in implementations that support multiple base ISAs. The effective XLEN in M-mode, MXLEN, is given by the setting of MXL, or has a fixed value if misa is zero. The MXL field is always set to the widest supported ISA variant at reset.

Table 8.2.1: Encoding of MXL field in misa

MXL	XLEN
1	32
2	64
3	128

The misa CSR is MXLEN bits wide. If the value read from misa is nonzero, field MXL of that value always denotes the current MXLEN. If a write to misa causes MXLEN to change, the position of MXL moves to the most-significant two bits of misa at the new width.

The Extensions field encodes the presence of the standard extensions, with a single bit per letter of the alphabet (bit 0 encodes presence of extension “A” , bit 1 encodes presence of extension “B”, through to bit 25 which encodes “Z”). The “I” bit will be set for RV32I, RV64I, RV128I base ISAs, and the “E” bit will be set for RV32E. The Extensions field is a **WARL** field that can contain writable bits where the implementation allows the supported ISA to be modified. At reset, the Extensions field shall contain the maximal set of supported extensions, and I shall be selected over E if both are available. When a standard extension is disabled by clearing its bit in misa, the instructions and CSRs defined or modified by the extension revert to their defined or reserved behaviors as if the extension is not implemented. The design of the RV128I base ISA is not yet complete, and while much of the remainder of this specification is expected to apply to RV128, this version of the document focuses only on RV32 and RV64. The “U” and “S” bits will be set if there is support for user and supervisor modes respectively. The “X” bit will be set if there are any non-standard extensions.

Table 8.2.2: Encoding of Extensions field in misa. All bits that are reserved for future use must return zero when read.

Bit	Character	Description
0	A	Atomic extension
1	B	Tentatively reserved for Bit-Manipulation extension
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E base ISA
5	F	Single-precision floating-point extension
6	G	Reserved
7	H	Hypervisor extension
8	I	RV32I/64I/128I base ISA
9	J	entatively reserved for Dynamically Translated Languages extension
10	K	Reserved
11	L	Reserved
12	M	Integer Multiply/Divide extension
13	N	Tentatively reserved for User-Level Interrupts extension
14	O	Reserved
15	P	Tentatively reserved for Packed-SIMD extension
16	Q	Quad-precision floating-point extension
17	R	Reserved
18	S	Supervisor mode implemented
19	T	Reserved

continues on next page

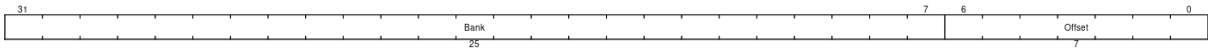
Table 8.2.2 – continued from previous page

Bit	Character	Description
20	U	User mode implemented
21	V	Tentatively reserved for Vector extension
22	W	Reserved
23	X	Non-standard extensions present
24	Y	Reserved
25	Z	Reserved

The “E” bit is read-only. Unless misa is all read-only zero, the “E” bit always reads as the complement of the “I” bit. An implementation that supports both RV32E and RV32I can select RV32E by clearing the “I” bit. If an ISA feature x depends on an ISA feature y, then attempting to enable feature x but disable feature y results in both features being disabled. For example, setting “F”=0 and “D”=1 results in both “F” and “D” being cleared. An implementation may impose additional constraints on the collective setting of two or more misa fields, in which case they function collectively as a single **WARL** field. An attempt to write an unsupported combination causes those bits to be set to some supported combination. Writing misa may increase IALIGN, e.g., by disabling the “C” extension. If an instruction that would write misa increases IALIGN, and the subsequent instruction’s address is not IALIGN-bit aligned, the write to misa is suppressed, leaving misa unchanged. When software enables an extension that was previously disabled, then all state uniquely associated with that extension is unspecified, unless otherwise specified by that extension.

Machine Vendor ID Register mvendorid

The mvendorid CSR is a 32-bit read-only register providing the JEDEC manufacturer ID of the provider of the core. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented or that this is a non-commercial implementation.



JEDEC manufacturer IDs are ordinarily encoded as a sequence of one-byte continuation codes 0x7f, terminated by a one-byte ID not equal to 0x7f, with an odd parity bit in the most-significant bit of each byte. mvendorid encodes the number of one-byte

continuation codes in the Bank field, and encodes the final byte in the Offset field, discarding the parity bit. For example, the JEDEC man- ufacturer ID 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x8a (twelve continuation codes followed by 0x8a) would be encoded in the mvendorid CSR as 0x60a. Mindgrove's VendorID is 0x7e6.

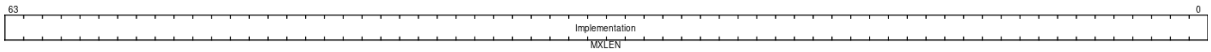
Machine Architecture ID Register marchid

The marchid CSR is an MXLEN-bit read-only register encoding the base microarchitecture of the hart. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented. The combination of mvendorid and marchid should uniquely identify the type of hart microarchitecture that is implemented.

Open-source project architecture IDs are allocated globally by RISC-V International, and have non-zero architecture IDs with a zero most-significant-bit (MSB). Commercial architecture IDs are allocated by each commercial vendor independently, but must have the MSB set and cannot contain zero in the remaining MXLEN-1 bits.

Machine Implementation ID Register mimpid

The mimpid CSR provides a unique encoding of the version of the processor implementation. This register must be readable in any implementation, but a value of 0 can be returned to indicate that the field is not implemented. The Implementation value should reflect the design of the RISC-V processor itself and not any surrounding system.

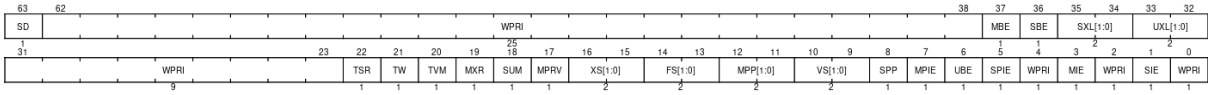


Hart ID Register mhartid

The mhartid CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero. Hart IDs must be unique within the execution environment.

Machine Status Registers (mstatus and mstatush)

The mstatus register is an MXLEN-bit read/write register formatted as shown in below. The mstatus register keeps track of and controls the hart's current operating state. A restricted view of mstatus appears as the sstatus register in the S-level ISA.



Privilege and Global Interrupt-Enable Stack in mstatus register

Global interrupt-enable bits, MIE and SIE, are provided for M-mode and S-mode respectively. These bits are primarily used to guarantee atomicity with respect to interrupt handlers in the current privilege mode.

When a hart is executing in privilege mode x, interrupts are globally enabled when x IE=1 and globally disabled when x IE=0. Interrupts for lower-privilege modes, w<x, are always globally disabled regardless of the setting of any global w IE bit for the lower-privilege mode. Interrupts for higher-privilege modes, y>x, are always globally enabled regardless of the setting of the global yIE bit for the higher-privilege mode. Higher-privilege-level code can use separate per-interrupt enable bits to disable selected higher-privilege-mode interrupts before ceding control to a lower-privilege mode.

To support nested traps, each privilege mode x that can respond to interrupts has a two-level stack of interrupt-enable bits and privilege modes. x PIE holds the value of the interrupt-enable bit active prior to the trap, and x PP holds the previous privilege mode. The x PP fields can only hold privilege modes up to x, so MPP is two bits wide and SPP is one bit wide. When a trap is taken from privilege mode y into privilege mode x, x PIE is set to the value of x IE; x IE is set to 0; and x PP is set to y.

An MRET or SRET instruction is used to return from a trap in M-mode or S-mode respectively. When executing an x RET instruction, supposing x PP holds the value y, x IE is set to x PIE; the privilege mode is changed to y; x PIE is set to 1; and x PP is set to the least-privileged supported mode (U if U-mode is implemented, else M). If x PP $\neq$ M, x RET also sets MPRV=0. x PP fields are **WARL** fields that can hold only privilege mode x and any implemented privilege mode lower than x. If privilege mode x is not implemented, then x PP must be read-only 0.

Base ISA Control in mstatus Register

For RV64 systems, the SXL and UXL fields are **WARL** fields that control the value of XLEN for S-mode and U-mode, respectively. The encoding of these fields is the same as the MXL field of misa. The effective XLEN in S-mode and U-mode are termed SXLEN and UXLEN, respectively. For RV32 systems, the SXL and UXL fields do not exist, and SXLEN=32 and UXLEN=32. For RV64 systems, if S-mode is not supported, then SXL is read-only zero. Otherwise, it is a **WARL** field that encodes the current value of SXLEN. In particular, an implementation may make SXL be a read-only field whose value always ensures that SXLEN=MXLEN. For RV64 systems, if U-mode is not supported, then UXL is read-only zero. Otherwise, it is a **WARL** field that encodes the current value of UXLEN. In particular, an implementation may make UXL be a read-only field whose value always ensures that UXLEN=MXLEN or UXLEN=SXLEN. Whenever XLEN in any mode is set to a value less than the widest supported XLEN, all operations must ignore source operand register bits above the configured XLEN, and must sign-extend results to fill the entire widest supported XLEN in the destination register. Similarly, pc bits above XLEN are ignored, and when the pc is written, it is sign-extended to fill the widest supported XLEN.

If MXLEN is changed from 32 to a wider width, each of mstatus fields SXL and UXL, if not restricted to a single value, gets the value corresponding to the widest supported width not wider than the new MXLEN.

Memory Privilege in mstatus Register

The MPRV (Modify PRiVilege) bit modifies the effective privilege mode, i.e., the privilege level at which loads and stores execute. When MPRV=0, loads and stores behave as normal, using the translation and protection mechanisms of the current privilege mode. When MPRV=1, load and store memory addresses are translated and protected, and endianness is applied, as though the current privilege mode were set to MPP. Instruction address-translation and protection are unaffected by the setting of MPRV. MPRV is read-only 0 if U-mode is not supported. An MRET or SRET instruction that changes the privilege mode to a mode less privileged than M also sets MPRV=0. The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect. MXR is read-only 0 if S-mode is not supported.

The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory ac-

cesses to pages that are accessible by U-mode (U=1) will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect. Note that, while SUM is ordinarily ignored when not executing in S-mode, it is in effect when MPRV=1 and MPP=S. SUM is read-only 0 if S-mode is not supported or if satp.MODE is read-only 0. The MXR and SUM mechanisms only affect the interpretation of permissions encoded in page-table entries. In particular, they have no impact on whether access-fault exceptions are raised due to PMAs or PMP.

Endianness Control in mstatus and mstatush Registers

The MBE, SBE, and UBE bits in mstatus and mstatush are **WARL** fields that control the endianness of memory accesses other than instruction fetches. Instruction fetches are always little- endian. MBE controls whether non-instruction-fetch memory accesses made from M-mode (assuming mstatus.MPRV=0) are little-endian (MBE=0) or big-endian (MBE=1). If S-mode is not supported, SBE is read-only 0. Otherwise, SBE controls whether explicit load and store memory accesses made from S-mode are little-endian (SBE=0) or big-endian (SBE=1). If U-mode is not supported, UBE is read-only 0. Otherwise, UBE controls whether explicit load and store memory accesses made from U-mode are little-endian (UBE=0) or big-endian (UBE=1). For implicit accesses to supervisor-level memory management data structures, such as page tables, endianness is always controlled by SBE. Since changing SBE alters the implementation's interpre- tation of these data structures, if any such data structures remain in use across a change to SBE, M-mode software must follow such a change to SBE by executing an SFENCE.VMA instruction with rs1=x0 and rs2=x0.

If S-mode is supported, an implementation may make SBE be a read-only copy of MBE. If U-mode is supported, an implementation may make UBE be a read-only copy of either MBE or SBE.

Virtualization Support in mstatus Register

The TVM (Trap Virtual Memory) bit is a **WARL** field that supports intercepting supervisor virtual-memory management operations. When TVM=1, attempts to read or write the satp CSR or execute an SFENCE.VMA or SINVAL.VMA instruction while executing in S-mode will raise an illegal instruction exception. When TVM=0, these operations are permitted in S-mode. TVM is read-only 0 when S-mode is not supported.

The TW (Timeout Wait) bit is a **WARL** field that supports intercepting the WFI instruction . When TW=0, the WFI instruction may execute in lower privilege modes when not prevented for some other reason. When TW=1, then if WFI is executed in any less-

privileged mode, and it does not complete within an implementation-specific, bounded time limit, the WFI instruction causes an illegal instruction exception. The time limit may always be 0, in which case WFI always causes an illegal instruction exception in less-privileged modes when TW=1. TW is read-only 0 when there are no modes less privileged than M.

When S-mode is implemented, then executing WFI in U-mode causes an illegal instruction excep- tion, unless it completes within an implementation-specific, bounded time limit. A future revision of this specification might add a feature that allows S-mode to selectively permit WFI in U-mode. Such a feature would only be active when TW=0. The TSR (Trap SRET) bit is a **WARL** field that supports intercepting the supervisor exception return instruction, SRET. When TSR=1, attempts to execute SRET while executing in S-mode will raise an illegal instruction exception. When TSR=0, this operation is permitted in S-mode. TSR is read-only 0 when S-mode is not supported.

Extension Context Status in mstatus Register

Supporting substantial extensions is one of the primary goals of RISC-V, and hence we define a standard interface to allow unchanged privileged-mode code, particularly a supervisor-level OS, to support arbitrary user-mode state extensions.

The FS[1:0] and VS[1:0] **WARL** fields and the XS[1:0] read-only field are used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit and any other user-mode extensions respectively. The FS field encodes the status of the floating-point unit state, including the floating-point registers f0–f31 and the CSRs fcsr, frm, and fflags. The VS field encodes the status of the vector extension state, including the vector registers v0–v31 and the CSRs vcsr, vxrm, vxsat, vstart, vl, vtype, and vlenb. The XS field encodes the status of additional user-mode extensions and associated state. These fields can be checked by a Context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

The FS, VS, and XS fields use the same status encoding as shown below, with the four possible status values being Off, Initial, Clean, and Dirty.

Table 8.2.3: Encoding of FS[1:0], VS[1:0], and XS[1:0] status fields.

Status	FS and VS Meaning	Description
0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

If the F extension is implemented, the FS field shall not be read-only zero. If neither the F extension nor S-mode is implemented, then FS is read-only zero. If S-mode is implemented but the F extension is not, FS may optionally be read-only zero.

If the v registers are implemented, the VS field shall not be read-only zero. If neither the v registers nor S-mode is implemented, then VS is read-only zero. If S-mode is implemented but the v registers are not, VS may optionally be read-only zero. In systems without additional user extensions requiring new state, the XS field is read-only zero. Every additional extension with state provides a CSR field that encodes the equivalent of the XS states.

The SD bit is a read-only bit that summarizes whether either the FS, VS, or XS fields signal the presence of some dirty state that will require saving extended user context to memory. If FS, XS, and VS are all read-only zero, then SD is also always zero. When an extension's status is set to Off, any instruction that attempts to read or write the corresponding state will cause an illegal instruction exception. When the status is Initial, the corre- sponding state should have an initial constant value. When the status is Clean, the corresponding state is potentially different from the initial value, but matches the last value stored on a context swap. When the status is Dirty, the corresponding state has potentially been modified since the last context save.

During a context save, the responsible privileged code need only write out the corresponding state if its status is Dirty, and can then reset the extension's status to Clean. During a context restore, the context need only be loaded from memory if the status is Clean (it should never be Dirty at restore). If the status is Initial, the context must be set to an initial constant value on context restore to avoid a security hole, but this can be done without accessing memory. For example, the floating-point registers can all be initialized to the immediate value 0. The FS and XS fields are read by the privileged code before saving the context. The FS field is set directly by privileged code when resuming a user context, while the XS field is set indirectly by writing to the status register

of the individual extensions. The status fields will also be updated during execution of instructions, regardless of privilege mode. Extensions to the user-mode ISA often include additional user-mode state, and this state can be considerably larger than the base integer registers. The extensions might only be used for some applications, or might only be needed for short phases within a single application. To improve performance, the user-mode extension can define additional instructions to allow user-mode software to return the unit to an initial state or even to turn off the unit. For example, a coprocessor might require to be configured before use and can be "unconfigured" after use. The unconfigured state would be represented as the Initial state for context save. If the same application remains running between the unconfigure and the next configure (which would set status to Dirty), there is no need to actually reinitialize the state at the unconfigure instruction, as all state is local to the user process, i.e., the Initial state may only cause the coprocessor state to be initialized to a constant value at context restore, not at every unconfigure. Executing a user-mode instruction to disable a unit and place it into the Off state will cause an illegal instruction exception to be raised if any subsequent instruction tries to use the unit before it is turned back on. A user-mode instruction to turn a unit on must also ensure the unit's state is properly initialized, as the unit might have been used by another context meantime. Changing the setting of FS has no effect on the contents of the floating-point register state. In particular, setting FS=Off does not destroy the state, nor does setting FS=Initial clear the contents. Similarly, the setting of VS has no effect on the contents of the vector register state. Other extensions, however, might not preserve state when set to Off. Implementations may choose to track the dirtiness of the floating-point register state imprecisely by reporting the state to be dirty even when it has not been modified. On some implementations, some instructions that do not mutate the floating-point state may cause the state to transition from Initial or Clean to Dirty. On other implementations, dirtiness might not be tracked at all, in which case the valid FS states are Off and Dirty, and an attempt to set FS to Initial or Clean causes it to be set to Dirty.

If an instruction explicitly or implicitly writes a floating-point register or the fcsr but does not alter its contents, and FS=Initial or FS=Clean, it is implementation-defined whether FS transitions to Dirty.

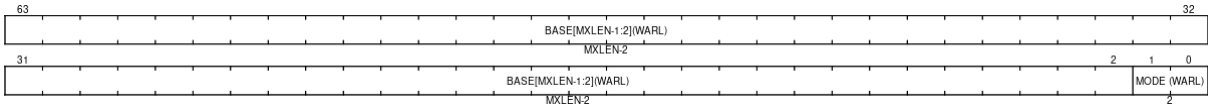
Implementations may choose to track the dirtiness of the vector register state in an analogous imprecise fashion, including possibly setting VS to Dirty when software attempts to set VS=Initial or VS=Clean. When VS=Initial or VS=Clean, it is implementation-defined whether an instruction that writes a vector register or vector CSR but does not alter its contents causes VS to transition to Dirty.

Standard privileged instructions to initialize, save, and restore extension state are provided to insulate privileged code from details of the added extension state by treating the state as an opaque object.

The XS field provides a summary of all added extension state, but additional microarchitectural bits might be maintained in the extension to further reduce context save and restore overhead. The SD bit is read-only and is set when either the FS, VS, or XS bits encode a Dirty state (i.e., SD=((FS==11) OR (XS==11) OR (VS==11))). This allows privileged code to quickly determine when no additional context save is required beyond the integer register set and PC. The floating-point unit state is always initialized, saved, and restored using standard instructions (F, D, and/or Q), and privileged code must be aware of FLEN to determine the appropriate space to reserve for each f register. Machine and Supervisor modes share a single copy of the FS, VS, and XS bits. Supervisor-level software normally uses the FS, VS, and XS bits directly to record the status with respect to the supervisor-level saved context. Machine-level software must be more conservative in saving and restoring the extension state in their corresponding version of the context.

Machine Trap-Vector Base-Address Register (mtvec)

The mtvec register is an MXLEN-bit **WARL** read/write register that holds trap vector configura- tion, consisting of a vector base address (BASE) and a vector mode (MODE).



The mtvec register must always be implemented, but can contain a read-only value. If mtvec is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field.

Table 8.2.4: Encoding of mtvec MODE field.

Value	Name	Description
0	Direct	All exceptions set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
>=2	NIL	Reserved

The encoding of the MODE field is shown above. When MODE=Direct, all traps into machine mode cause the pc to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the pc to be

set to the address in the BASE field, whereas interrupts cause the pc to be set to the address in the BASE field plus four times the interrupt cause number. For example, a machine-mode timer interrupt causes the pc to be set to BASE+0x1c.

An implementation may have different alignment constraints for different modes. In particular, MODE=Vectored may have stricter alignment constraints than MODE=Direct.

Machine Trap Delegation Registers (medeleg and mideleg)

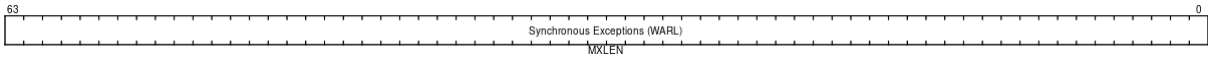
By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the MRET instruction. To increase performance, implementations can provide individual read/write bits within medeleg and mideleg to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level. The machine exception delegation register (medeleg) and machine interrupt delegation register (mideleg) are MXLEN-bit read/write registers. In systems with S-mode, the medeleg and mideleg registers must exist, and setting a bit in medeleg or mideleg will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S- mode trap handler. In systems without S-mode, the medeleg and mideleg registers should not exist.

When a trap is delegated to S-mode, the scause register is written with the trap cause; the sepc register is written with the virtual address of the instruction that took the trap; the stval register is written with an exception-specific datum; the SPP field of mstatus is written with the active privilege mode at the time of the trap; the SPIE field of mstatus is written with the value of the SIE field at the time of the trap; and the SIE field of mstatus is cleared. The mcause, mepc, and mtval registers and the MPP and MPIE fields of mstatus are not written.

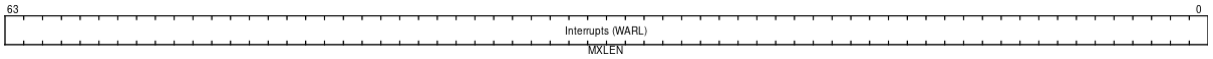
An implementation can choose to subset the delegatable traps, with the supported delegatable bits found by writing one to every bit location, then reading back the value in medeleg or mideleg to see which bit positions hold a one. An implementation shall not have any bits of medeleg be read-only one, i.e., any synchronous trap that can be delegated must support not being delegated. Similarly, an implementation shall not fix as read-only one any bits of mideleg corresponding to machine-level interrupts (but may do so for lower-level interrupts).

Traps never transition from a more-privileged mode to a less-privileged mode. For example, if M- mode has delegated illegal instruction exceptions to S-mode, and M-mode software later executes an illegal instruction, the trap is taken in M-mode, rather than being delegated to S-mode. By contrast, traps may be taken horizontally. Using the same example, if M-mode has delegated illegal instruction exceptions to S-mode, and S-mode software later executes an illegal instruction, the trap is taken in S-mode. Del-

egated interrupts result in the interrupt being masked at the delegator privilege level. For example, if the supervisor timer interrupt (STI) is delegated to S-mode by setting mideleg[5], STIs will not be taken when executing in M-mode. By contrast, if mideleg[5] is clear, STIs can be taken in any mode and regardless of current mode will transfer control to M-mode.



medeleg has a bit position allocated for every synchronous exception, with the index of the bit position equal to the value returned in the mcause register (i.e., setting bit 8 allows user-mode environment calls to be delegated to a lower-privilege trap handler).



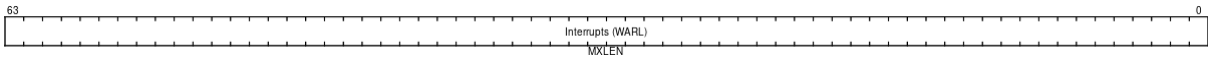
mideleg holds trap delegation bits for individual interrupts, with the layout of bits matching those in the mip register (i.e., STIP interrupt delegation control is located in bit 5). For exceptions that cannot occur in less privileged modes, the corresponding medeleg bits should be read-only zero. In particular, medeleg[11] is read-only zero.

Machine Interrupt Registers (mip and mie)

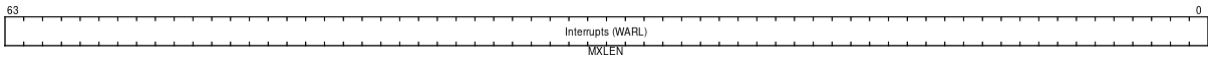
The mip register is an MXLEN-bit read/write register containing information on pending interrupts, while mie is the corresponding MXLEN-bit read/write register containing interrupt enable bits.

Interrupt cause number i corresponds with bit i inboth mip and mie. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform or custom use.

Machine Interrupt-Pending Register (mip):



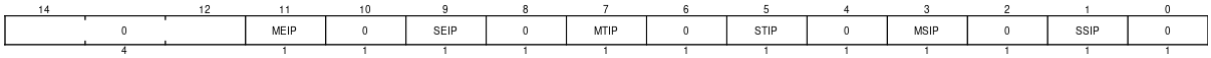
Machine Interrupt-Enable Register (mie):



An interrupt i will trap to M-mode (causing the privilege mode to change to M-mode) if all of the following are true: (a) either the current privilege mode is M and the MIE bit in the mstatus register is set, or the current privilege mode has less privilege than M-mode; (b) bit i is set in both mip and mie; and (c) if register mideleg exists, bit i is not set in mideleg. These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in mip, and must also be evaluated immediately following the execution of an x RET instruction or an explicit write to a CSR on which these interrupt trap conditions ex-

pressly depend (including mip, mie, mstatus, and mideleg). Interrupts to M-mode take priority over any interrupts to lower privilege modes. Each individual bit in register mip may be writable or may be read-only. When bit i in mip is writable, a pending interrupt i can be cleared by writing 0 to this bit. If interrupt i can become pending but bit i in mip is read-only, the implementation must provide some other mechanism for clearing the pending interrupt. A bit in mie must be writable if the corresponding interrupt can ever become pending. Bits of mie that are not writable must be read-only zero.

Standard portion (bits 15:0) of mip:



Standard portion (bits 15:0) of mip:

Bits mip.MEIP and mie.MEIE are the interrupt-pending and interrupt-enable bits for machine- level external interrupts. MEIP is read-only in mip, and is set and cleared by a platform-specific interrupt controller. Bits mip.MTIP and mie.MTIE are the interrupt-pending and interrupt-enable bits for machine timer interrupts. MTIP is read-only in mip, and is cleared by writing to the memory-mapped machine-mode timer compare register. Bits mip.MSIP and mie.MSIE are the interrupt-pending and interrupt-enable bits for machine- level software interrupts. MSIP is read-only in mip, and is written by accesses to memory-mapped control registers, which are used by remote harts to provide machine-level interprocessor interrupts. A hart can write its own MSIP bit using the same memory-mapped control register. If a system has only one hart, or if a platform standard supports the delivery of machine-level interprocessor interrupts through external interrupts (MEI) instead, then mip.MSIP and mie.MSIE may both be read-only zeros. If supervisor mode is not implemented, bits SEIP, STIP, and SSIP of mip and SEIE, STIE, and SSIE of mie are read-only zeros. If supervisor mode is implemented, bits mip.SEIP and mie.SEIE are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. SEIP is writable in mip, and may be written by M-mode software to indicate to S-mode that an external interrupt is pending. Addi- tionally, the platform-level interrupt controller may generate supervisor-level external interrupts. Supervisor-level external interrupts are made pending based on the logical-OR of the software- writable SEIP bit and the signal from the external interrupt controller. When mip is read with a CSR instruction, the value of the SEIP bit returned in the rd destination register is the logical- OR of the software-writable bit and the interrupt signal from the interrupt controller, but the signal from the interrupt controller is not used to calculate the value written to SEIP. Only the software-writable SEIP bit participates in the read-modify-write sequence of a CSRRS or CSRRC instruction.

If supervisor mode is implemented, bits mip.STIP and mie.STIE are the interrupt-

pending and interrupt-enable bits for supervisor-level timer interrupts. STIP is writable in mip, and may be written by M-mode software to deliver timer interrupts to S-mode. If supervisor mode is implemented, bits mip.SSIP and mie.SSIE are the interrupt-pending and interrupt-enable bits for supervisor-level software interrupts. SSIP is writable in mip and may also be set to 1 by a platform-specific interrupt controller. Multiple simultaneous interrupts destined for M-mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI.

Restricted views of the mip and mie registers appear as the sip and sie registers for supervisor level. If an interrupt is delegated to S-mode by setting a bit in the mideleg register, it becomes visible in the sip register and is maskable using the sie register. Otherwise, the corresponding bits in sip and sie are read-only zero.

Hardware Performance Monitor

M-mode includes a basic hardware performance-monitoring facility. The mcycle CSR counts the number of clock cycles executed by the processor core on which the hart is running. The minstret CSR counts the number of instructions the hart has retired. The mcycle and minstret registers have 64-bit precision on all RV32 and RV64 systems. The counter registers have an arbitrary value after the hart is reset, and can be written with a given value. Any CSR write takes effect after the writing instruction has otherwise completed. The mcycle CSR may be shared between harts on the same core, in which case writes to mcycle will be visible to those harts. The platform should provide a mechanism to indicate which harts share an mcycle CSR. The hardware performance monitor includes 29 additional 64-bit event counters, mhpmcounter3– mhpmcounter31. The event selector CSRs, mhpmevent3–mhpmevent31, are MXLEN-bit **WARL** registers that control which event causes the corresponding counter to increment. The meaning of these events is defined by the platform, but event 0 is defined to mean “no event.” All counters should be implemented, but a legal implementation is to make both the counter and its correspond- ing event selector be read-only 0.

To be done

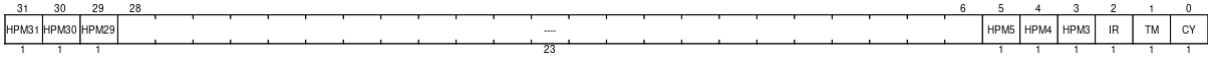
When MXLEN=32, reads of the mcycle, minstret, and mhpmcountern CSRs return bits 31–0 of the corresponding counter, and writes change only bits 31–0; reads of the mcycleh, minstreth, and mhpmcounternh CSRs return bits 63–32 of the corresponding counter, and writes change only bits 63–32.

To be done

Machine Counter-Enable Register (mcounteren)

The counter-enable register mcounteren is a 32-bit register that controls the availability of the hardware performance-monitoring counters to the next-lowest privileged mode.

Counter-enable register (mcounteren).:



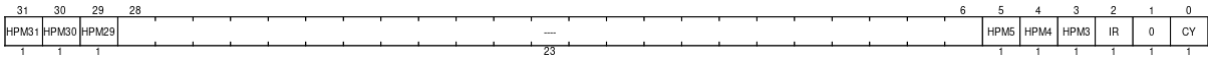
The settings in this register only control accessibility. The act of reading or writing this register does not affect the underlying counters, which continue to increment even when not accessible. When the CY, TM, IR, or HPMn bit in the mcounteren register is clear, attempts to read the cycle, time, instret, or hpmcountern register while executing in S-mode or U-mode will cause an illegal instruction exception. When one of these bits is set, access to the corresponding register is permitted in the next implemented privilege mode (S-mode if implemented, otherwise U-mode).

The cycle, instret, and hpmcountern CSRs are read-only shadows of mcycle, minstret, and mhpmcountern, respectively. The time CSR is a read-only shadow of the memory-mapped mtime register. Analogously, on RV32I the cycleh, instreth and hpmcounternh CSRs are read-only shadows of mcycleh, minstreth and mhpmcounternh, respectively. On RV32I the timeh CSR is a read-only shadow of the upper 32 bits of the memory-mapped mtime register, while time shadows only the lower 32 bits of mtime.

In systems with U-mode, the mcounteren must be implemented, but all fields are WARL and may be read-only zero, indicating reads to the corresponding counter will cause an illegal instruction exception when executing in a less-privileged mode. In systems without U-mode, the mcounteren register should not exist.

Machine Counter-Inhibit CSR (mcountinhibit)

Counter-inhibit register mcountinhibit:



The counter-inhibit register mcountinhibit is a 32-bit WARL register that controls which of the hardware performance-monitoring counters increment. The settings in this register only control whether the counters increment; their accessibility is not affected by the setting of this register. When the CY, IR, or HPMn bit in the mcountinhibit register is clear, the cycle, instret, or hpmcountern register increments as usual. When the CY, IR, or HPMn bit is set, the corresponding counter does not increment. The mcycle CSR may be shared between harts on the same core, in which case the mcountinhibit.CY field is also shared between those harts, and so writes to mcountinhibit.CY will be vis-

ible to those harts.

If the mcountinhibit register is not implemented, the implementation behaves as though the register were set to zero.

Machine Scratch Register (mscratch)

The mscratch register is an MXLEN-bit read/write register dedicated for use by machine mode. Typically, it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler.

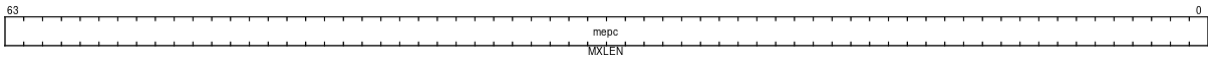
Machine-mode scratch register:

Machine Exception Program Counter (mepc)

mepc is an MXLEN-bit read/write register formatted as shown below. The low bit of mepc (mepc[0]) is always zero. On implementations that support only IALIGN=32, the two low bits (mepc[1:0]) are always zero. If an implementation allows IALIGN to be either 16 or 32 (by changing CSR misa, for example), then, whenever IALIGN=32, bit mepc[1] is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the MRET instruction. Though masked, mepc[1] remains writable when IALIGN=32. mepc is a WARL register that must be able to hold all valid virtual addresses. It need not be capable of holding all possible invalid addresses. Prior to writing mepc, implementations may convert an invalid address into some other invalid address that mepc is capable of holding.

When a trap is taken into M-mode, mepc is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, mepc is never written by the implementation, though it may be explicitly written by software.

Machine exception program counter register:

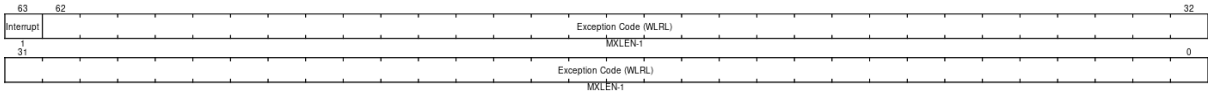


Machine Cause Register (mcause)

The mcause register is an MXLEN-bit read-write register formatted as shown below. When a trap is taken into M-mode, mcause is written with a code indicating the event that caused the trap. Otherwise, mcause is never written by the implementation, though it may be explicitly written by software. The Interrupt bit in the mcause register is set

if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. The Exception Code is a WLRL field, so is only guaranteed to hold supported exception codes.

Machine Cause register mcause:



Note that load and load-reserved instructions generate load exceptions, whereas store, store- conditional, and AMO instructions generate store/AMO exceptions.

If an instruction may raise multiple synchronous exceptions, the decreasing priority order of Ta- ble below indicates which exception is taken and reported in mcause. The priority of any custom synchronous exceptions is implementation-defined.

Table 8.2.5: Machine cause register (mcause) values after trap.

Interrupt	Exception Code	Description
1	0	Reserved
1	1	Supervisor software interrupt
1	2	Reserved
1	3	Machine software interrupt
1	4	Reserved
1	5	Supervisor timer interrupt
1	6	Reserved
1	7	Machine timer interrupt
1	8	Reserved
1	9	Supervisor external interrupt
1	10	Reserved
1	11	Machine external interrupt
1	12 - 15	Reserved
1	>=16	Designated for platform use
0	0	Instruction address misaligned

continues on next page

Table 8.2.5 – continued from previous page

Interrupt	Exception Code	Description
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	16 - 23	Reserved
0	24 - 31	Designated for custom use
0	32 - 47	Reserved
0	48 - 63	Designated for custom use
0	>= 64	Reserved

Table 8.2.6: Synchronous exception priority in decreasing priority order.

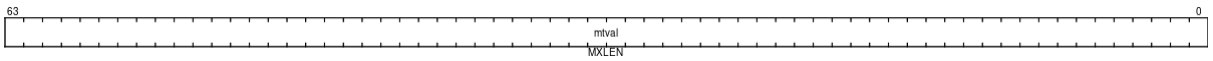
Priority	Exception Code	Description
Highest	3	Instruction address breakpoint
	12 , 1	During instruction address translation: First encountered page fault or access fault
	1	With physical address for instruction: Instruction access fault
	2	Illegal instruction
	0	Instruction address misaligned
	8, 9, 11	Environment call
	3	Environment break
	3	Load/store/AMO address breakpoint
	4, 6	Optionally: Load/Store/AMO address breakpoint
	13, 15, 5, 7	During address translation for an explicit memory access: First encountered page fault or access fault
	5, 7	With physical address for an explicit memory access: Load/store/AMO access fault
Lowest	4, 6	If not higher priority: Load/store/AMO address misaligned

When a virtual address is translated into a physical address, the address translation algorithm determines what specific exception may be raised. Load/store/AMO address-misaligned exceptions may have either higher or lower priority than load/store/AMO page-fault and access-fault exceptions.

Machine Trap Value Register (mtval)

The mtval register is an MXLEN-bit read-write register formatted as shown below. When a trap is taken into M-mode, mtval is either set to zero or written with exception-specific information to assist software in handling the trap. Otherwise, mtval is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which excep- tions must set mtval informatively and which may unconditionally set it to zero. If the hardware platform specifies that no exceptions set mtval to a nonzero value, then mtval is read-only zero. If mtval is written with a nonzero value when a breakpoint, address-misaligned, access-fault, or page-fault exception occurs on an instruction fetch, load, or store, then mtval will contain the faulting virtual address.

Machine Trap Value register:



If mtval is written with a nonzero value when a misaligned load or store causes an access-fault or page-fault exception, then mtval will contain the virtual address of the portion of the access that caused the fault. If mtval is written with a nonzero value when an instruction access-fault or page-fault exception occurs on a system with variable-length instructions, then mtval will contain the virtual address of the portion of the instruction that caused the fault, while mepc will point to the beginning of the instruction. The mtval register can optionally also be used to return the faulting instruction bits on an illegal instruction exception (mepc points to the faulting instruction in memory). If mtval is written with a nonzero value when an illegal-instruction exception occurs, then mtval will contain the shortest of:

- the actual faulting instruction
- the first ILEN bits of the faulting instruction
- the first MXLEN bits of the faulting instruction

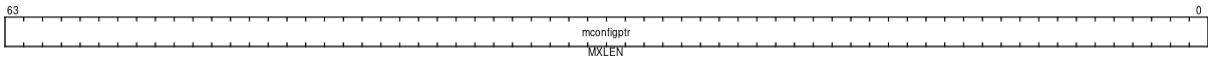
The value loaded into mtval on an illegal-instruction exception is right-justified and all unused upper bits are cleared to zero.

For other traps, mtval is set to zero, but a future standard may redefine mtval's setting for other traps. If mtval is not read-only zero, it is a WARL register that must be able to hold all valid virtual addresses and the value zero. It need not be capable of holding all possible invalid addresses. Prior to writing mtval, implementations may convert an invalid address into some other invalid address that mtval is capable of holding. If the feature to return the faulting instruction bits is implemented, mtval must also be able to hold all values less than 2N , where N is the smaller of MXLEN and ILEN.

Machine Configuration Pointer Register (mconfigptr)

mconfigptr is an MXLEN-bit read-only CSR, formatted as shown below, that holds the physical address of a configuration data structure. Software can traverse this data structure to discover information about the harts, the platform, and their configuration

Machine Configuration Pointer register.:

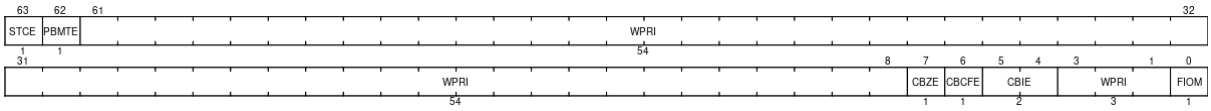


The pointer alignment in bits must be no smaller than the greatest supported MXLEN: i.e., if the greatest supported MXLEN is 8 × n, then mconfigptr[log2 n-1:0] must be zero. mconfigptr must be implemented, but it may be zero to indicate the configuration data structure does not exist or that an alternative mechanism must be used to locate it.

Machine Environment Configuration Registers (menvcfg and menvcfgh)

The menvcfg CSR is an MXLEN-bit read/write register, formatted for MXLEN=64 as shown below, that controls certain characteristics of the execution environment for modes less privileged than M.

Machine environment configuration register (menvcfg) for MXLEN=64:



If bit FIOM (Fence of I/O implies Memory) is set to one in menvcfg, FENCE instructions executed in modes less privileged than M are modified so the requirement to order accesses to device I/O implies also the requirement to order main memory accesses. Similarly, for modes less privileged than M when FIOM=1, if an atomic instruction that accesses a region ordered as device I/O has its aq and/or rl bit set, then that instruction is ordered as though it accesses both device I/O and memory. If S-mode is not supported, or if satp.MODE is read-only zero (always Bare), the implementation may make FIOM read-only zero.

Table 8.2.7: Modified interpretation of FENCE predecessor and successor sets for modes less privi-leged than M when FIOM=1.

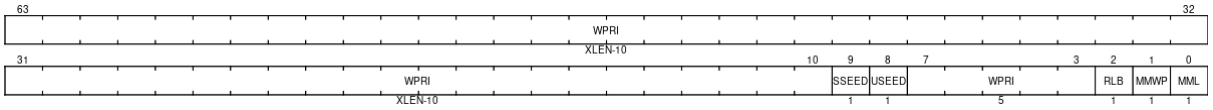
Instruction bit	Meaning when set
PI	Predecessor device input and memory reads (PR implied)
PO	Predecessor device output and memory writes (PW implied)
SI	Successor device input and memory reads (SR implied)
SO	Successor device output and memory writes (SW implied)

The PBMTE bit controls whether the Svpbmt extension is available for use in S-mode and G-stage address translation (i.e., for page tables pointed to by satp or hgatp). When PBMTE=1, Svpbmt is available for S-mode and G-stage address translation. When PBMTE=0, the implementation behaves as though Svpbmt were not implemented. If Svpbmt is not implemented, PBMTE is read- only zero. Furthermore, for implementations with the hypervisor extension, henvcfg.PBMTE is read-only zero if menvcfg.PBMTE is zero. The definition of the STCE field will be furnished by the forthcoming Sstc extension. Its allocation within menvcfg may change prior to the ratification of that extension. The definition of the CBZE field will be furnished by the forthcoming Zicboz extension. Its alloca- tion within menvcfg may change prior to the ratification of that extension.

The definitions of the CBCFE and CBIE fields will be furnished by the forthcoming Zicbom ex- tension. Their allocations within menvcfg may change prior to the ratification of that extension. When MXLEN=32, menvcfg contains the same fields as bits 31:0 of menvcfg when MXLEN=64. Additionally, when MXLEN=32, menvcfgh is a 32-bit read/write register that contains the same fields as bits 63:32 of menvcfg when MXLEN=64. Register menvcfgh does not exist when MXLEN=64. If U-mode is not supported, then registers menvcfg and menvcfgh do not exist.

Machine Security Configuration Register (mseccfg)

mseccfg is an optional MXLEN-bit read/write register, formatted as shown below, that controls security features. When MXLEN=32 only, mseccfgh is a 32-bit read/write register that contains the same fields as mseccfg bits 63:32 when MXLEN=64.



The definitions of the SSEED and USEED fields will be furnished by the forthcoming entropy- source extension, Zkr. Their allocations within mseccfg may change prior to the ratification of that extension. The definitions of the RLB, MMWP, and MML fields will be furnished by the forthcoming PMP- enhancement extension, Smepmp. Their allocations within mseccfg may change prior to the rati- fication of that extension.

8.3 Supervisor-Level ISA, Version 1.12

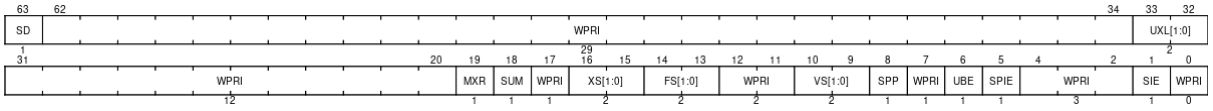
8.3.1 Supervisor-Level CSRs

A number of CSRs are provided for the supervisor.

Supervisor Status Register(sstatus)

The sstatus register is an SXLEN-bit read/write register. The sstatus register keeps track of the processor's current operating state. The SPP bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, SPP is set to 0 if the trap originated from user mode, or 1 otherwise. When an SRET instruction (see Section 3.3.2) is executed to return from the trap handler, the privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0.

Supervisor-mode status register (sstatus) when SXLEN=64:



privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0. The SIE bit enables or disables all interrupts in supervisor mode. When SIE is clear, interrupts are not taken while in supervisor mode. When the hart is running in user-mode, the value in SIE is ignored, and supervisor-level interrupts are enabled. The supervisor can disable individual interrupt sources using the sie CSR. The SPIE bit indicates whether supervisor interrupts were enabled prior to trapping into supervisor mode. When a trap is taken into supervisor mode, SPIE is set to SIE, and SIE is set to 0. When an SRET instruction is executed, SIE is set to SPIE, then SPIE is set to 1. The sstatus register is a subset of the mstatus register.

Base ISA Control in sstatus Register

The UXL field controls the value of XLEN for U-mode, termed UXLEN, which may differ from the value of XLEN for S-mode, termed SXLEN. The encoding of UXL is the same as that of the MXL field of misa. When SXLEN=32, the UXL field does not exist, and UXLEN=32. When SXLEN=64, it is a WARL field that encodes the current value of UXLEN. In particular, an implementation may make UXL be a read-only field whose value always ensures that UXLEN=SXLEN.

If UXLEN $\neq$ SXLEN, instructions executed in the narrower mode must ignore source register operand bits above the configured XLEN, and must sign-extend results to fill the widest supported XLEN in the destination register. If UXLEN < SXLEN, user-mode instruction-fetch addresses and load and store effective addresses are taken modulo 2UXLEN. For example, when UXLEN=32 and SXLEN=64, user-mode memory accesses reference the lowest 4 GiB of the address space.

Memory Privilege in sstatus Register

The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect. The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1) will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect, nor when executing in U-mode. Note that S-mode can never execute instructions from user pages, regardless of the state of SUM. SUM is read-only 0 if satp.MODE is read-only 0.

Endianness Control in sstatus Register

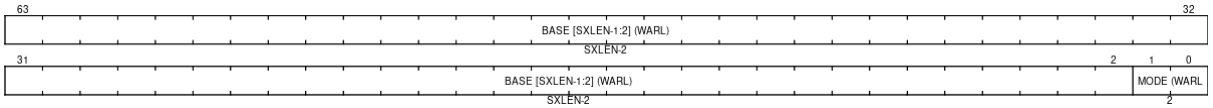
The UBE bit is a WARL field that controls the endianness of explicit memory accesses made from U-mode, which may differ from the endianness of memory accesses in S-mode. An implementation may make UBE be a read-only field that always specifies the same endianness as for S-mode. UBE controls whether explicit load and store memory accesses made from U-mode are little-endian (UBE=0) or big-endian (UBE=1). UBE has no effect on instruction fetches, which are implicit memory accesses that are always little-endian.

For implicit accesses to supervisor-level memory management data structures, such as page tables, S-mode endianness always applies and UBE is ignored.

Supervisor Trap Vector Base Address Register (stvec)

The stvec register is an SXLEN-bit read/write register that holds trap vector configuration, con- sisting of a vector base address (BASE) and a vector mode (MODE).

Supervisor trap vector base address register (stvec):



The BASE field in stvec is a WARL field that can hold any valid virtual or physical address, subject to the following alignment constraints: the address must be 4-byte aligned, and MODE settings other than Direct might impose additional alignment constraints on the value in the BASE field.

Table 8.3.1: Encoding of stvec MODE field.

Value	Name	Description
0	Direct	All exceptions set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
>=2	NIL	Reserved

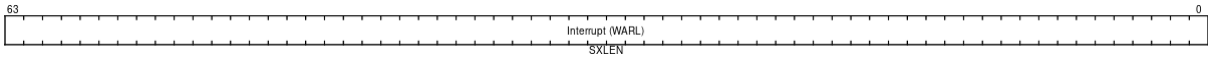
The encoding of the MODE field is shown above. When MODE=Direct, all traps into supervisor mode cause the pc to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into supervisor mode cause the pc to be set to the address in the BASE field, whereas interrupts cause the pc to be set to the address in the BASE field plus four times the interrupt cause number. For example, a supervisor-mode timer interrupt causes the pc to be set to BASE+0x14. Setting MODE=Vectored may impose a stricter alignment constraint on BASE.

Supervisor Interrupt Registers (sip and sie)

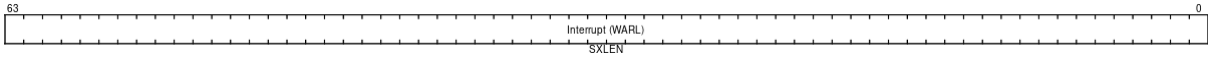
The sip register is an SXLEN-bit read/write register containing information on pending interrupts, while sie is the corresponding SXLEN-bit read/write register containing in-

terrupt enable bits. Interrupt cause number i (as reported in CSR scause, Section 4.1.8) corresponds with bit i in both sip and sie. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform or custom use.

Supervisor interrupt-pending register (sip):

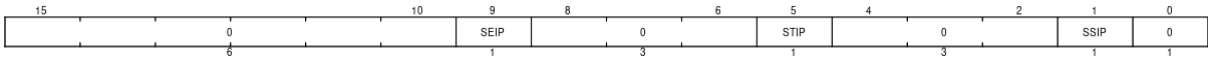


Supervisor interrupt-enable register (sie):

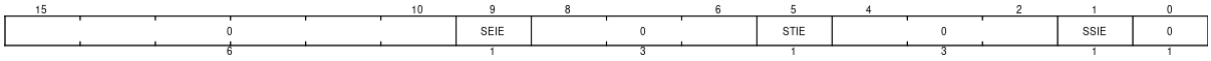


An interrupt i will trap to S-mode if both of the following are true: (a) either the current privilege mode is S and the SIE bit in the sstatus register is set, or the current privilege mode has less privilege than S-mode; and (b) bit i is set in both sip and sie. These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in sip, and must also be evaluated immediately following the execution of an SRET instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend (including sip, sie and sstatus). Interrupts to S-mode take priority over any interrupts to lower privilege modes. Each individual bit in register sip may be writable or may be read-only. When bit i in sip is writable, a pending interrupt i can be cleared by writing 0 to this bit. If interrupt i can become pending but bit i in sip is read-only, the implementation must provide some other mechanism for clearing the pending interrupt (which may involve a call to the execution environment). A bit in sie must be writable if the corresponding interrupt can ever become pending. Bits of sie that are not writable are read-only zero.

Standard portion (bits 15:0) of sip:



Standard portion (bits 15:0) of sie:



Bits sip.SEIP and sie.SEIE are the interrupt-pending and interrupt-enable bits for supervisor- level external interrupts. If implemented, SEIP is read-only in sip, and is set and cleared by the execution environment, typically through a platform-specific interrupt controller.

Bits sip.STIP and sie.STIE are the interrupt-pending and interrupt-enable bits for supervisor- level timer interrupts. If implemented, STIP is read-only in sip, and is set and cleared by the execution environment. Bits sip.SSIP and sie.SSIE are the interrupt-pending and interrupt-enable bits for supervisor- level software interrupts. If implemented, SSIP is writable in sip and may also be set to 1 by a platform-specific interrupt controller.

Each standard interrupt type (SEI, STI, or SSI) may not be implemented, in which case the corresponding interrupt-pending and interrupt-enable bits are read-only zeros. All bits in sip and sie are WARL fields. The implemented interrupts may be found by writing one to every bit location in sie, then reading back to see which bit positions hold a one.

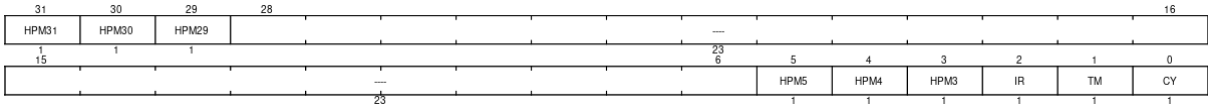
Multiple simultaneous interrupts destined for supervisor mode are handled in the following decreas- ing priority order: SEI, SSI, STI.

Supervisor Timers and Performance Counters

Supervisor software uses the same hardware performance monitoring facility as user-mode software, including the time, cycle, and instret CSRs. The implementation should provide a mechanism to modify the counter values. The implementation must provide a facility for scheduling timer interrupts in terms of the real-time counter, time.

Counter-Enable Register (scounteren)

Counter-enable register (scounteren):



The counter-enable register scounteren is a 32-bit register that controls the availability of the hardware performance monitoring counters to U-mode.

When the CY, TM, IR, or HPMn bit in the scounteren register is clear, attempts to read the cycle, time, instret, or hpmcountern register while executing in U-mode will cause an illegal instruction exception. When one of these bits is set, access to the corresponding register is permitted. scounteren must be implemented. However, any of the bits may be read-only zero, indicating reads to the corresponding counter will cause an exception when executing in U-mode. Hence, they are effectively WARL fields.

Supervisor Scratch Register (sscratch)

The sscratch register is an SXLEN-bit read/write register, dedicated for use by the supervisor. Typically, sscratch is used to hold a pointer to the hart-local supervisor context while the hart is executing user code. At the beginning of a trap handler, sscratch is swapped with a user register to provide an initial working register.

Supervisor Scratch Register:

Supervisor Exception Program Counter (sepc)

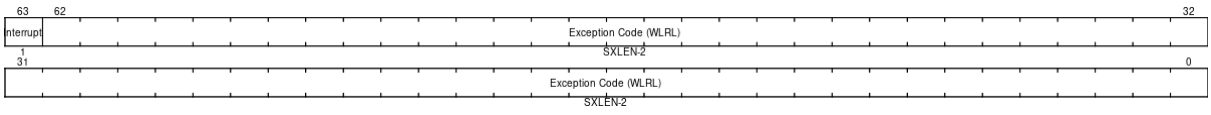
sepc is an SXLEN-bit read/write register. The low bit of sepc (sepc[0]) is always zero. On implementations that support only IALIGN=32, the two low bits (sepc[1:0]) are always zero. If an implementation allows IALIGN to be either 16 or 32 (by changing CSR misa, for example), then, whenever IALIGN=32, bit sepc[1] is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the SRET instruction. Though masked, sepc[1] remains writable when IALIGN=32. sepc is a WARL register that must be able to hold all valid virtual addresses. It need not be capable of holding all possible invalid addresses. Prior to writing sepc, implementations may convert an invalid address into some other invalid address that sepc is capable of holding. When a trap is taken into S-mode, sepc is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, sepc is never written by the implementation, though it may be explicitly written by software.

Supervisor exception program counter register:

Supervisor Cause Register (scause)

The scause register is an SXLEN-bit read-write register. Whena trap is taken into S-mode, scause is written with a code indicating the event that caused the trap. Otherwise, scause is never written by the implementation, though it may be explicitly written by software. The Interrupt bit in the scause register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. The Exception Code is a WLRL field. It is required to hold the values 0–31 (i.e., bits 4–0 must be implemented), but otherwise it is only guaranteed to hold supported exception codes.

Supervisor Cause register scause:



Supervisor Trap Value (stval) Register

The stval register is an SXLEN-bit read-write register. When a trap is taken into S-mode, stval is written with exception-specific information to assist software in handling the trap. Otherwise, stval is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set

stval informatively and which may unconditionally set it to zero. If stval is written with a nonzero value when a breakpoint, address-misaligned, access-fault, or page-fault exception occurs on an instruction fetch, load, or store, then stval will contain the faulting virtual address.

Supervisor Trap Value register:

If stval is written with a nonzero value when a misaligned load or store causes an access-fault or page-fault exception, then stval will contain the virtual address of the portion of the access that caused the fault. If stval is written with a nonzero value when an instruction access-fault or page-fault exception occurs on a system with variable-length instructions, then stval will contain the virtual address of the portion of the instruction that caused the fault, while sepc will point to the beginning of the instruction.

Table 8.3.2: Supervisor cause register (scause) values after trap. Synchronous exception priorities are given

Interrupt	Exception Code	Description
1	0	Reserved
1	1	Supervisor software interrupt
1	2 - 4	Reserved
1	5	Supervisor timer interrupt
1	6 - 8	Reserved
1	9	Supervisor external interrupt
1	10 - 15	Reserved
1	>=16	Designated for platform use
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault

continues on next page

Table 8.3.2 – continued from previous page

Interrupt	Exception Code	Description
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10 - 11	Reserved
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	16 - 23	Reserved
0	24 - 31	Designated for custom use
0	32 - 47	Reserved
0	48 - 63	Designated for custom use
0	>= 64	Reserved

The stval register can optionally also be used to return the faulting instruction bits on an illegal instruction exception (sepc points to the faulting instruction in memory). If stval is written with a nonzero value when an illegal-instruction exception occurs, then stval will contain the shortest of:

- the actual faulting instruction
- the first ILEN bits of the faulting instruction
- the first SXLEN bits of the faulting instruction

The value loaded into stval on an illegal-instruction exception is right-justified and all unused upper bits are cleared to zero.

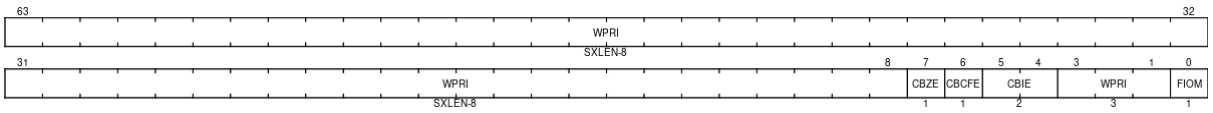
For other traps, stval is set to zero, but a future standard may redefine stval's setting for other traps. stval is a WARL register that must be able to hold all valid virtual addresses and the value 0. It need not be capable of holding all possible invalid addresses. Prior to writing stval, implemen- tations may convert an invalid address into some other invalid address that stval is capable of holding. If the feature to return the faulting instruction

bits is implemented, stval must also be able to hold all values less than 2N , where N is the smaller of SXLEN and ILEN.

Supervisor Environment Configuration Register (senvcfg)

The senvcfg CSR is an SXLEN-bit read/write register, that controls certain characteristics of the U-mode execution environment.

Supervisor Trap Value register:



f bit FIOM (Fence of I/O implies Memory) is set to one in senvcfg, FENCE instructions executed in U-mode are modified so the requirement to order accesses to device I/O implies also the re- quirement to order main memory accesses. Similarly, for U-mode when FIOM=1, if an atomic instruction that accesses a region ordered as device I/O has its aq and/or rl bit set, then that instruction is ordered as though it accesses both device I/O and memory. If satp.MODE is read-only zero (always Bare), the implementation may make FIOM read-only zero.

Table 8.3.3: Modified interpretation of FENCE predecessor and successor sets in U-mode when FIOM=1.

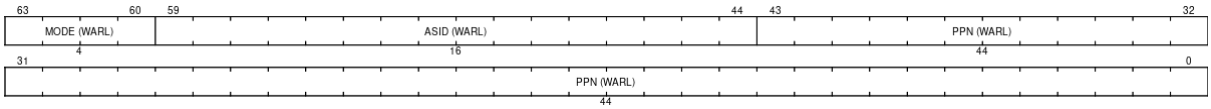
Instruction bit	Meaning when set
PI	Predecessor device input and memory reads (PR implied)
PO	Predecessor device output and memory writes (PW implied)
SI	Successor device input and memory reads (SR implied)
SO	Successor device output and memory writes (SW implied)

The definition of the CBZE field will be furnished by the forthcoming Zicboz extension. Its alloca- tion within senvcfg may change prior to the ratification of that extension. The definitions of the CBCFE and CBIE fields will be furnished by the forthcoming Zicbom ex- tension. Their allocations within senvcfg may change prior to the ratification of that extension.

Supervisor Address Translation and Protection (satp) Register

The satp register is an SXLEN-bit read/write register, which controls supervisor-mode address translation and protection. This register holds the physical page number (PPN) of the root page table, i.e., its supervisor physical address divided by 4 KiB; an address space identifier (ASID), which facilitates address-translation fences on a per-address-space basis; and the MODE field, which selects the current address-translation scheme.

Supervisor address translation and protection register satp when SXLEN=64, for MODE values Bare, Sv39, Sv48, and Sv57:



When MODE=Bare, supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme described in Section 3.7. To select MODE=Bare, software must write zero to the remaining fields of satp (bits 30–0 when SXLEN=32, or bits 59–0 when SXLEN=64). Attempting to select MODE=Bare with a nonzero pattern in the remaining fields has an unspecified effect on the value that the remaining fields assume and an unspecified effect on address translation and protection behavior. When SXLEN=32, the satp encodings corresponding to MODE=Bare and ASID[8:7]=3 are des- ignated for custom use, whereas the encodings corresponding to MODE=Bare and ASID[8:7] $\neq 3$ are reserved for future standard use. When SXLEN=64, all satp encodings corresponding to MODE=Bare are reserved for future standard use.

When SXLEN=32, the only other valid setting for MODE is Sv32, a paged virtual-memory scheme described in Section 4.3. When SXLEN=64, three paged virtual-memory schemes are defined: Sv39, Sv48, and Sv57, de- scribed in Sections 4.4, 4.5, and 4.6, respectively. One additional scheme, Sv64, will be defined in a later version of this specification. The remaining MODE settings are reserved for future use and may define different interpretations of the other fields in satp. Implementations are not required to support all MODE settings, and if satp is written with an unsupported MODE, the entire write has no effect; no fields in satp are modified. The number of ASID bits is unspecified and may be zero. The number of implemented ASID bits, termed ASIDLEN, may be determined by writing one to every bit position in the ASID field, then reading back the value in satp to see which bit positions in the ASID field hold a one. The least-significant bits of ASID are implemented first: that is, if ASIDLEN > 0, ASID[ASIDLEN-1:0] is writable. The maximal value of ASIDLEN, termed ASIDMAX, is 9 for Sv32 or 16 for Sv39, Sv48, and Sv57.

Table 8.3.4: Encoding of satp MODE field.

Value	Name	Description
0	Bare	No translation or protection
1 - 7	NIL	Reserved for standard use
8	Sv39	Page-based 39-bit virtual addressing
9	Sv48	Page-based 48-bit virtual addressing
10	Sv57	Page-based 57-bit virtual addressing
11	Sv64	Reserved for page-based 64-bit virtual addressing.
12 - 13	NIL	Reserved for standard use
14 - 15	NIL	Designated for custom use

The satp register is considered active when the effective privilege mode is S-mode or U-mode. Executions of the address-translation algorithm may only begin using a given value of satp when satp is active.

Note that writing satp does not imply any ordering constraints between page-table updates and subsequent address translations, nor does it imply any invalidation of address-translation caches. If the new address space's page tables have been modified, or if an ASID is reused, it may be necessary to execute an SFENCE.VMA instruction (see Section 4.2.1) after, or in some cases before, writing satp.

For further more details refer RISC-V Documentation

8.4 Physical Memory Protection (PMP)

Physical Memory Protection (PMP) provides a hardware mechanism to control access to physical memory by software executing on a hart. PMP allows privileged software to define memory regions with configurable read, write, and execute permissions. These permissions are selectively enforced by hardware based on the configured PMP entries and privilege mode. The S2401 supports a minimum PMP granularity of 8 bytes and provides 8 configurable PMP entries.

8.4.1 Address Registers

The S2401 provides eight PMP address registers, pmpaddr0 through pmpaddr7. Each pmpaddr register is 64 bits wide. Bits 55:3 encode the physical address, corresponding to a 56-bit physical address space. Since the least significant three bits represent address offsets from 0 to 7, the minimum memory region that PMP can protect is 8 bytes; therefore, these bits are not stored in the register. The encoded address specifies the start or end of the protected region depending on the address-matching mode (NAPOT or TOR). The physical address must be right-shifted by three bits before being written to the corresponding pmpaddrx register. The PMP address register must be configured before configuring the associated bit field in the configuration register (pmpcfg0).

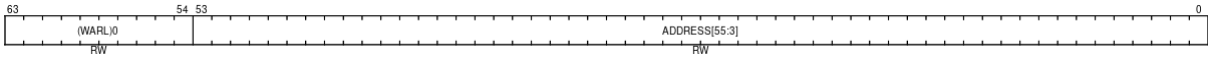


Fig. 8.4.1: pmpaddrx

Note

The pmpcfg0 and pmpaddrx registers can be configured only in machine mode.

8.4.2 Configuration Register

The S2401 provides a single PMP configuration register, pmpcfg0, which is 64 bits wide and contains eight 8-bit fields. Each field configures the permissions and attributes of the corresponding PMP address register. i.e:

Table 8.4.1: pmpcfg0 bit field mapping

Bit field	PMP Entry
pmpcfg0 [7:0]	pmpaddr0
pmpcfg0 [15:8]	pmpaddr1
pmpcfg0 [23:16]	pmpaddr2
pmpcfg0 [31:24]	pmpaddr3
pmpcfg0 [39:32]	pmpaddr4
pmpcfg0 [47:40]	pmpaddr5

continues on next page

Table 8.4.1 – continued from previous page

Bit field	PMP Entry
pmpcfg0 [55:48]	pmpaddr6
pmpcfg0 [63:56]	pmpaddr7

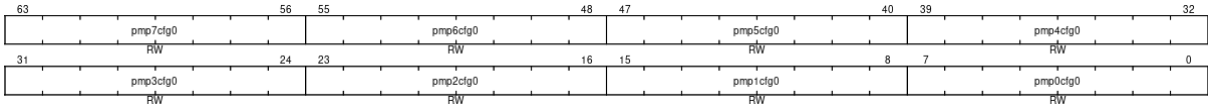


Fig. 8.4.2: pmpcfg0

The description of each 8-bit field is given below:

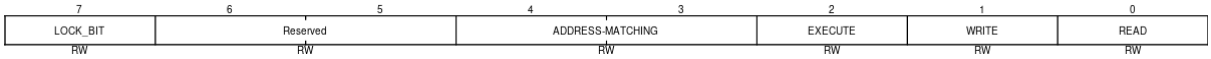


Fig. 8.4.3: pmpxcfg0

Table 8.4.2: pmpxcfg

Bits	Field name	Permis-sion	Description
[1:0]	Read	RW	When this bit is cleared, read access to the memory region defined by the corresponding pmpaddrX entry is not permitted, and a read access to this region triggers a load access fault. When this bit is set, read access to the memory region defined by the corresponding pmpaddrX entry is permitted.

continues on next page

Table 8.4.2 – continued from previous page

Bits	Field name	Permission	Description
[2:1]	Write	RW	When this bit is cleared, write access to the memory region defined by the corresponding pmpaddrX entry is not permitted, and a write access to this region triggers a store access fault. When this bit is set, write access to the memory region defined by the corresponding pmpaddrX entry is permitted.
[3:2]	Execute	RW	When this bit is cleared, instruction fetches from the memory region defined by the corresponding pmpaddrX entry are not permitted, and an instruction fetch from this region triggers an instruction access fault. When this bit is set, instruction fetches from the memory region defined by the corresponding pmpaddrX entry are permitted.
[5:3]	Address matching	RW	• **00** - OFF(protection is off). • **01** - TOR(Top of Range). • **11** - NAPOT(Naturally Alligned Power of Two)
[7:5]	Reserved	RW	Reserved for future use.
[8:7]	Lock-bit	RW	When cleared, the PMP configuration applies only to user mode. When set, the PMP configuration also applies to machine mode.

Note

If PMP entry x is locked, writes to pmpxcfg and pmpaddrx are ignored. It cannot be modified until the hart is reset.

The S2401 supports only TOR and NAPOT address matching modes.

- **OFF** - PMP is off.
- **TOR (Top Of Range)** - It protects between two address given by pmpaddrx regis-

ters.

  - If pmpaddr0 is configured as TOR then PMP protects from very start addr eg: 0x00000000 - pmpaddr0.
  - If pmpaddr1 is configured as TOR then PMP protects from last address protected by pmpaddr0 to pmpaddr1.
  - If pmpaddr2 is configured as TOR then PMP protects from last address protected by pmpaddr1 to pmpaddr2.

- **NAPOT (Naturally Aligned Power Of Two)** - It protects minimum of 8 bytes from the given address and the byte length can be increased by appending the 1's from the 4th bit of address from LSB with a stopping 0. The LSB 3 bits must be zero since the address is right shifted 3 times. NAPOT configuration is given below:

Table 8.4.3: NAPOT address encoding

Address(binary)	Region size(in bytes)
yyyy...yyy0000	8
yyyy...yy01000	16
yyyy...y011000	32
yyyy...0111000	64
yy01...1111000	(2^XLEN)
y011...1111000	(2^XLEN+1)
0111...1111000	(2^XLEN+2)
1111...1111000	(2^XLEN+3)

- Example: If pmpaddrx = 0x80000000 is configured with the NAPOT then it protects from 0x80000000 to 0x80000007 which is 8 bytes.
- If pmpaddrx = 0x80000008 is configured with the NAPOT, since from the LSB the 4th bit is 1 and the stoping zero is at 5th bit, it protects from 0x80000000 to 0x8000000f which is 16 bytes.
- If pmpaddrx = 0x80000018 is configured with the NAPOT, since from the LSB the 4th bit and 5th bit is 1 and the stoping zero is at 6th bit, it protects from 0x80000000 to 0x8000001f which is 32 bytes.

Note

In TOR mode, ensure the previous entry is configured (*pmp(x-1)cfg0* and *pmpaddr(x-1)*) before configuring *pmpxcfg0* and *pmpaddrx*, as protection starts from the end of the address protected by the previous entry.

8.5 Performance monitors

The Performance Monitor is a hardware feature used to analyze and optimize system performance by tracking various metrics during execution. It provides insights into critical aspects like cache behavior (hits, misses, fill buffer usage), pipeline stalls, branch prediction accuracy, and arithmetic operation counts. These metrics help in identifying bottlenecks, understanding resource utilization, and improving the efficiency of tasks like matrix multiplication or other compute-intensive operations.

8.5.1 Performance Counters and Metrics

The number of counters and metrics depends on the hardware architecture and its performance monitoring capabilities.

8.5.2 Counters

Each field in the structs represents a hardware counter that tracks a specific performance metric. Counting all fields:

Cache_perf Counters

The *Cache_perf* structure contains counters related to instruction and data cache performance. Below is a list of each counter with a brief description:

- **Ifbhit**: Tracks the number of instruction cache fill buffer hits. This occurs when data requested is already in the fill buffer, avoiding a full cache miss.
- **Ifbrelease**: Counts the number of times the instruction cache fill buffer is released after a request is serviced.

- **Imiss**: Counts the number of instruction cache misses. A miss happens when the requested instruction is not in the cache, leading to a memory fetch.
- **Inc_access**: Tracks the number of non-cached memory accesses for instructions.
- **Iaccess**: Counts the total number of instruction cache accesses.
- **Dread_access**: Tracks the number of data read accesses to the cache.
- **Dwrite_access**: Counts the number of data write accesses to the cache.
- **Datomic_access**: Tracks the number of atomic data accesses to the cache. Atomic operations involve both read and write actions.
- **Dnc_read_access**: Tracks the number of non-cached read accesses to data memory.
- **Dnc_write_access**: Tracks the number of non-cached write accesses to data memory.
- **Dread_miss**: Counts the number of cache misses during data read operations.
- **Dwrite_miss**: Counts the number of cache misses during data write operations.
- **Datomic_miss**: Tracks the number of cache misses during atomic data operations.
- **Dread_fbhit**: Tracks the number of data read fill buffer hits. This occurs when requested data is in the fill buffer, avoiding a full cache miss.
- **Dwrite_fbhit**: Tracks the number of data write fill buffer hits.
- **Datomic_fbhit**: Tracks the number of fill buffer hits during atomic data operations.
- **Dfbrelease**: Counts the number of times the data cache fill buffer is released after a request is serviced.
- **Dline_evictions**: Tracks the number of data cache line evictions. Evictions occur when a cache line is replaced to accommodate new data.
- **Itlb_miss**: Counts the number of instruction TLB (Translation Lookaside Buffer) misses. This happens when a virtual address for an instruction is not found in the TLB.
- **Dtlb_miss**: Counts the number of data TLB misses. This occurs when a virtual address for data is not found in the TLB.

Stalls_perf Counters

The *Stalls_perf* structure tracks pipeline stalls, which occur when the processor cannot proceed with instruction execution.

- **rawstalls**: Counts the number of Read-After-Write (RAW) stalls. These stalls occur when an instruction depends on the result of a previous instruction that hasn't been completed yet.
- **exestalls**: Counts the number of execution stalls. These happen when the execution pipeline is stalled due to resource contention, memory latency, or other delays.

Branch_perf Counters

The *Branch_perf* structure monitors branch instruction performance, focusing on prediction and execution behavior.

- **misprediction**: Tracks the number of branch mispredictions. This occurs when the processor's branch predictor incorrectly predicts the target of a branch instruction, causing a pipeline flush.
- **jumps**: Counts the number of jump instructions executed. Jump instructions alter the program counter (PC) to a new location in the code.
- **branches**: Tracks the total number of branch instructions executed, including conditional and unconditional branches.

Arith_perf Counters

The *Arith_perf* structure monitors arithmetic operations performed by the processor.

- **floats**: Counts the number of floating-point arithmetic operations executed.
- **muldiv**: Tracks the number of multiplication and division operations executed. These operations are typically more resource-intensive than simple additions or subtractions.

8.5.3 Metrics

Metrics refer to what these counters measure. Combining all fields, the metrics include:

- **Cache Metrics**: Cache hits, misses, fill buffer usage, access types (read, write, atomic), TLB misses, and cache evictions.
- **Pipeline Stalls**: Read-after-write (RAW) stalls and execution stalls.
- **Branch Metrics**: Mispredictions, jumps, and branch instruction counts.
- **Arithmetic Metrics**: Floating-point operations and multiply/divide operations.

Chapter 9. Debuggers and Itracing

9.1 RISC-V Debugger

9.1.1 Introduction

The **RISC-V Debugger** provides software developers and embedded engineers with a low-level interface to inspect, control, and debug RISC-V-based systems. This document describes the debugger's architecture, supported debugging features, register mappings, and command usage.

The debugger operates using **GDB** and **OpenOCD**, communicating with the target RISC-V processor via JTAG or SWD.

The RISC-V Debugger consists of the following components:

- **Debug Host**: A computer running GDB/OpenOCD.
- **Debug Translator**: Software such as OpenOCD that translates GDB commands into low-level debug transport commands.
- **Debug Transport Hardware**: A JTAG/SWD adapter that connects the Debug Host to the target platform.
- **Debug Transport Module (DTM)**: The first component inside the RISC-V platform, handling external debug commands.
- **Debug Module (DM)**: Manages debug operations for one or more harts.
- **Harts (Hardware Threads)**: Each RISC-V processor hart (hardware thread) is controlled by one DM.

Note

This implementation supports only one hart.

9.1.2 Register Map and Details

Table 9.1.1: Debugger Module Registers

Register Name	Address	Description
data0	0x04	Abstract data registers are used for storing data in debug operations. There are 16 Abstract data registers starting from 0 till 15 .
dmcontrol	0x0F	Controls the debug module, including selecting harts and enabling debug module.
dmstatus	0x11	Provides status of the debug module, such as authentication, availability, and halt status.
abstractcs	0x16	Controls and provides status for abstract command execution.
command	0x17	Executes an abstract command (e.g., register read/write, memory access).
abstractauto	0x18	Enables auto-execution for abstract commands to speed up debugging.
progbuf0	0x20	The program buffer is used to execute arbitrary instructions on the target. The Size of the Program Buffer in 32-bit words, valid sizes are 0 - 16.
sbcs	0x38	Controls system bus access and provides status for system bus operations.
sbaddress0	0x39	The lower 32 bits of the system bus address.
sbaddress1	0x3A	The upper 32 bits of the system bus address.
sbdata0	0x3C	The lower 32 bits of system bus data for read/write operations.
sbdata1	0x3D	The upper 32 bits of system bus data for read/write operations.
haltsum0	0x40	Indicates which harts are halted in a multi-hart debug environment.

data0 (Abstract Data 0) Register

data0 (**DATA0** to **DATA11**) is a 32-bit registers that holds data used by abstract commands for memory and register operations. We can read from or write to these registers during an abstract command. Use these registers for memory operations or other tasks in an abstract command. Avoid using them while the command is running, as it will set **CMDDERR** to 1 (busy).

Fig. 9.1.1: *data0*

dmcontrol (Debug Module Control) Register

The **dmcontrol** is a 32-bit register that manages and controls debugging in a RISC-V system. Used to stop, inspect, modify, and resume execution during development and debugging.

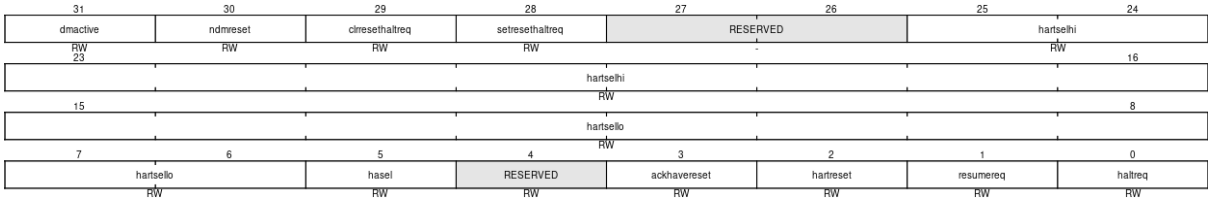


Fig. 9.1.2: *dmcontrol*

Table 9.1.2: Debug Module Control Register Field

Bits	Field Name	Access	Description
[0:0]	haltreq	RW	Writing 0 clears the halt request for all currently selected harts; writing 1 sets the halt request. Running harts will halt when their halt request is set.
[1:1]	resumereq	RW	Writing 1 causes currently selected harts to resume once if halted, and clears the resume ack bit for those harts. Ignored if haltreq is set.

continues on next page

Table 9.1.2 – continued from previous page

Bits	Field Name	Access	Description
[2:2]	hartreset	RW	Writing 1 asserts the reset for all currently selected harts. Must not change selected harts while this bit is set. Not implemented if it always reads 0.
[3:3]	ack-havereset	RW	Writing 1 clears the havereset status for any selected harts; has no effect if set to 0.
[4:4]	RESERVED	RW	Reserved.
[5:5]	hasel	RW	Selects the definition of currently selected harts. Set to 0 for single hart, or to 1 for multiple harts if hart array mask register is supported.
[15:6]	hartsello	RW	Low 10 bits of hartsel for DM-specific index of the selected hart.
[25:16]	hartselhi	RW	High 10 bits of hartsel for DM-specific index of the selected hart.
[27:26]	RESERVED	RW	Reserved for future use.
[28:28]	setresethaltreq	RW	Sets halt-on-reset request for all currently selected harts unless clrresethaltreq is set. Must clear manually; not implemented if hasresethaltreq is 0.
[29:29]	clrresethaltreq	RW	Clears halt-on-reset request for all currently selected harts.
[30:30]	ndmreset	RW	Controls reset signal to the system, asserting a system-wide reset. Writing 1 resets the system, writing 0 deasserts.

continues on next page

Table 9.1.2 – continued from previous page

Bits	Field Name	Access	Description
[31:31]	dmactive	RW	Serves as a reset signal for the Debug Module. Writing 0 resets the module state; writing 1 returns it to normal operation. Should be managed to prevent unexpected resets during debugging.

dmstatus (Debug Module Status) Register

The **dmstatus** is a 32-bit status register used to monitor the state of the RISC-V Debug Module and harts to determine whether debugging operations can proceed.

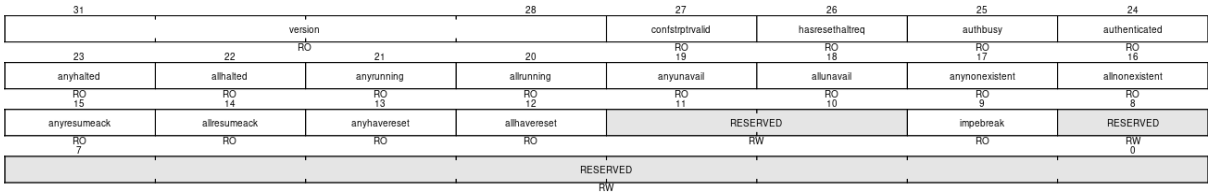


Fig. 9.1.3: *dmstatus*

Table 9.1.3: Debug Module Status Register Field

Bits	Field Name	Access	Description
[8:0]	RESERVED	RW	Reserved.
[9:9]	impebreak	RO	Indicates implicit *ebreak* support; must be 1 if *progbufsize* is 1.
[11:10]	RESERVED	RW	Reserved.
[12:12]	allhavereset	RO	This bit is set when all selected harts have been reset and not acknowledged.
[13:13]	anyhavereset	RO	This bit is set when any selected hart has been reset and not acknowledged.

continues on next page

Table 9.1.3 – continued from previous page

Bits	Field Name	Access	Description
[14:14]	allresumeack	RO	This bit is set when all selected harts acknowledge their last resume request.
[15;15]	anyresumeack	RO	This bit is set when any selected hart acknowledges its last resume request.
[16:16]	allnonexistent	RO	This bit is set when all selected harts do not exist.
[17:17]	anynonexistent	RO	This bit is set when any selected hart does not exist.
[18:18]	allunavail	RO	This bit is set when all selected harts are unavailable.
[19:19]	anyunavail	RO	This bit is set when any selected hart is unavailable.
[20:20]	allrunning	RO	This bit is set when all selected harts are running.
[21:21]	anyrunning	RO	This bit is set when any selected hart is running.
[22:22]	allhalted	RO	This bit is set when all selected harts are halted.
[23:23]	anyhalted	RO	This bit is set when any selected hart is halted.
[24:24]	authenticated	RO	This bit is set if the debugger is authenticated; otherwise, it must be 0.
[25:25]	authbusy	RO	This bit is set when the authentication module is busy; otherwise, it is 0.
[26:26]	hasresethaltreq	RO	This bit is set if halt-on-reset functionality is supported.
[27:27]	confstrptrvalid	RO	This bit is set if the configuration string pointer holds valid information.

continues on next page

Table 9.1.3 – continued from previous page

Bits	Field Name	Access	Description
[31:28]	version	RO	Indicates the version of the Debug Module with 2 for version 0.13.

abstractcs (Abstract Control And Status) Register

The **abstractcs** is a 32-bit register that controls and reports the status of abstract commands in the RISC-V Debug Module. It is used to monitor command execution, detect errors, and manage command buffers. Writing to this register while an abstract command is executing causes cmderr to be set to 1 (busy), when the command completes (busy becomes 0).

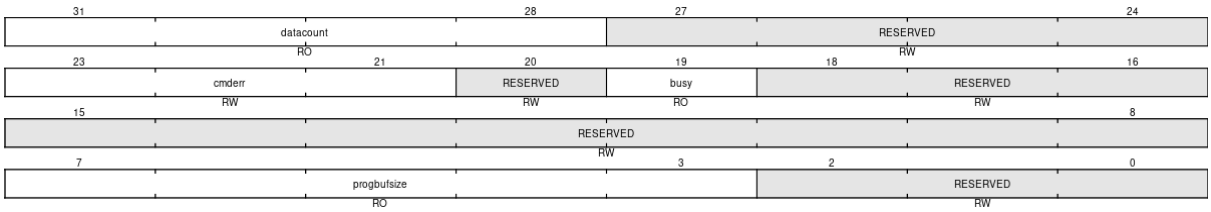


Fig. 9.1.4: *abstractcs*

Table 9.1.4: Abstract Control And Status Register Field

Bits	Field Name	Access	Description
[2:0]	RESERVED	RW	Reserved.
[7:3]	progbufsize	RO	Size of the Program Buffer in 32-bit words, valid sizes are 0 - 16.
[18:8]	RESERVED	RW	Reserved.

continues on next page

Table 9.1.4 – continued from previous page

Bits	Field Name	Access	Description
[19:19]	busy	RO	Set to 1 if an abstract command is currently being executed. This bit is set as soon as the command is written and is cleared only when the command completes.
[20:20]	RESERVED	RW	Reserved.
[23:21]	cmderr	RW	Indicates specific error codes for different failure conditions (e.g. busy, not supported, exception, halt/resume, bus, other) if an abstract command fails. It is a 'write-1-to-clear' register. Its value is valid when busy is 0.
[27:24]	RESERVED	RW	Reserved.
[31:28]	datacount	RO	Number of data registers implemented as part of the abstract command interface. Valid sizes are 1 - 12.

command (Abstract Command) Register

The **command** is a 32-bit register used to issue abstract commands in the RISC-V Debug Module.

The fields of the register are interpreted differently according to the value given in **cmdtype**:

- **cmdtype = 0 (Access Register Command)**

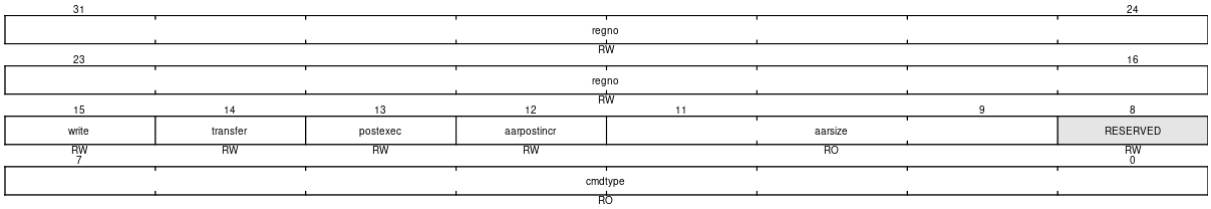


Fig. 9.1.5: *command*

Table 9.1.5: Abstract Command Register Field

Bits	Field Name	Attribute	Description
[7:0]	cmdtype	RO	Specifies the type of the abstract command, which defines its overall functionality, such as register access operations. This is set to 0 to indicate Access Register Command.
[8]	RESERVED	RW	Reserved.
[11:9]	aarsize	RO	Defines the size of the access being performed: - 0: 8-bit access - 1: 16-bit access - 2: 32-bit access - 3: 64-bit access - 4: 128-bit access
[12:12]	aarpostincr	RW	Controls auto-increment for the address after the access operation. If set to 1, the address is incremented automatically.
[13:13]	postexec	RW	If set to 1, the program buffer is executed after the completion of this command.
[14:14]	transfer	RW	Initiates a data transfer operation. This bit must be set to start transferring data to or from the specified address.
[15:15]	write	RW	Indicates the type of operation: - 1: Write operation - 0: Read operation.
[31:16]	regno	RW	Specifies the register number to access. This field is used to indicate which register is involved in the operation.

- **cmdtype = 1 (Quick Access Command)**

Used for fast access to a specific register, if implemented. Follows a simplified execution flow compared to `cmdtype = 0`.

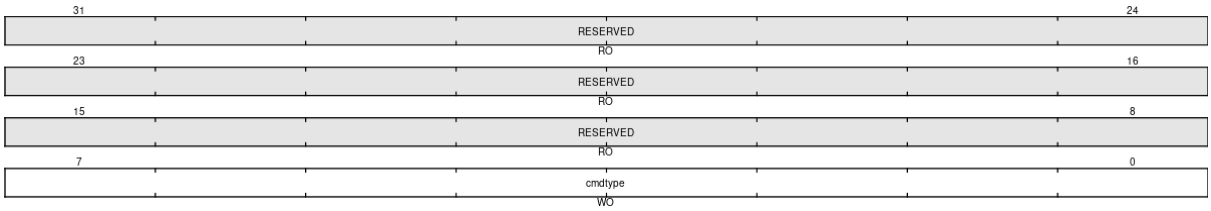


Fig. 9.1.6: *Quick Access*

Table 9.1.6: Quick Access Command Field

Bits	Field Name	Attribute	Description
[7:0]	cmdtype	WO	This field indicates that the command is a Quick Access command (set to 1).
[31:8]	RESERVED	RO	RESERVED for future use, must be set to zero.

- **cmdtype = 2 (Access Memory Command)**

  When `cmdtype = 2`, the fields of the **command** register are used for target-specific operations and are structured as follows:

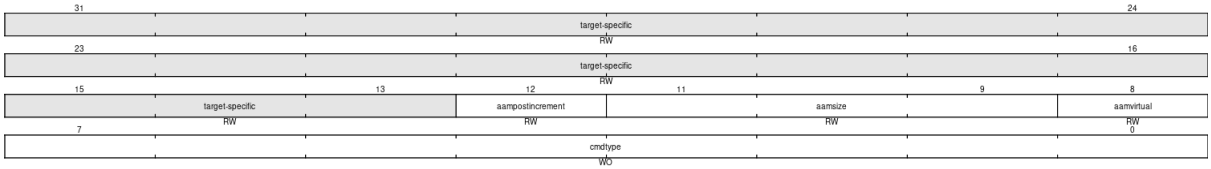


Fig. 9.1.7: *Access Memory Command*

Table 9.1.7: Access Memory Command Field (cmdtype = 2)

Bits	Field Name	Attribute	Description
[7:0]	cmdtype	WO	This field indicates that the command is an Access Memory Command (set to 2).
[8:8]	aamvirtual	RW	This field indicates the address type. If set, it indicates virtual addresses; else it indicates physical addresses, translated from M-mode with MPRV set.
[11:9]	aamsize	RW	Determines the size of the memory access: - 0: 8 bits, - 1: 16 bits, - 2: 32 bits, - 3: 64 bits, - 4: 128 bits.
[12:12]	aampostincrement	RW	If 1, increments arg1 by the number of bytes specified in aamsize after the access.
[31:13]	targetspecific	RW	Reserved for target-specific uses.

abstractauto (Abstract Command Auto Exec) Register

The **abstractauto** is a 32-bit register that enables automatic execution of abstract commands in the RISC-V Debug Module when specified events occur, such as hitting a breakpoint. Once an abstract command is configured, this register allows the command to be triggered automatically without manual intervention, and can be disabled to stop automatic execution.

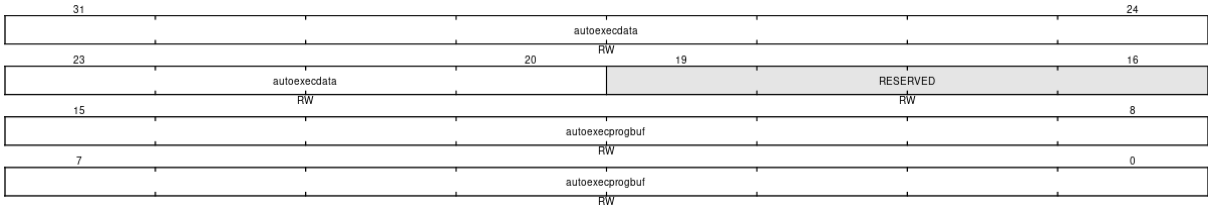


Fig. 9.1.8: *abstractauto*

Table 9.1.8: Abstract Command Autoexec Register Field

Bits	Field Name	Attribute	Description
[15:0]	autoexecprogbuf	RW	If set to 1, read/write accesses to the program buffer will re-execute the command in the **abstract command** register.
[19:16]	RESERVED	RW	Reserved.
[31:20]	autoexecdata	RW	If set to 1, read/write accesses to the data word will re-execute the command in the **abstract command** register.

progbuf0 (Program Buffer) Register

progbuf0 through **progbuf15** are 32-bit registers that provide write access to the optional program buffer.

The debugger may also be able to read from the program buffer through these registers. If reading is not supported, all reads return 0. Accessing these registers while an abstract command is executing sets **CMDDERR** to 1 (busy). Writing to them while busy has no effect. The debugger can write whatever program it likes (including jumps out of the Program Buffer), but the program must end with ebreak or c.ebreak.

An implementation may support an implicit ebreak that is executed when a hart runs off the end of the Program Buffer. This is indicated by impebreak.

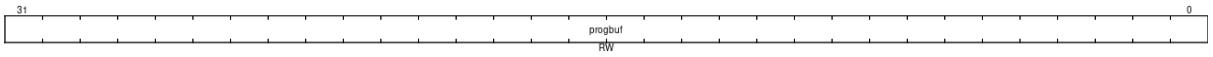


Fig. 9.1.9: *progbuf0*

sbcs (System Bus Access Control and Status) Register

The **sbcs** is a 32-bit register that reflects the status and control of system bus access operations

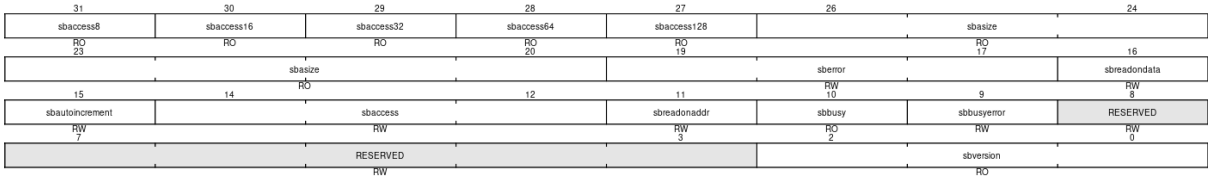


Fig. 9.1.10: *sbcs*

Table 9.1.9: SBCS Register Field

Bits	Field Name	Attribute	Description
[2:0]	sbversion	RO	1: The System Bus interface complies with version 0.13 of the specification.
[8:3]	RESERVED	RW	Reserved
[9:9]	sbbusyerror	RW	Set when new access is initiated while busy. Remains set until cleared by the debugger.
[10:10]	sbbusy	RO	Indicates the system bus master is busy. High during read/write requests and low after completion.
[11:11]	sbreadonaddr	RW	When 1, writing to sbaddress0 triggers a system bus read at the new address.
[14:12]	sbaccess	RW	Selects access size: 0=8-bit, 1=16-bit, 2=32-bit, 3=64-bit, 4=128-bit. Unsupported values set sberror to 4.
[15:15]	sbautoincrement	RW	When 1, sbaddress increments by access size after every system bus access.
[16:16]	sbreadondata	RW	When 1, reading from sbdata0 triggers a system bus read at the address.

continues on next page

Table 9.1.9 – continued from previous page

Bits	Field Name	Attribute	Description
[19:17]	sberror	RW	Set on system bus errors. Non-zero indicates errors (e.g., 1-timeout, 2-bad address, 3-alignment error, 4-unsupported size and 7-other errors.).
[26:20]	sbasize	RO	Width of system bus addresses in bits. 0 indicates no bus access support.
[27:27]	sbaccess128	RO	1 when 128-bit system bus accesses are supported.
[28:28]	sbaccess64	RO	1 when 64-bit system bus accesses are supported.
[29:29]	sbaccess32	RO	1 when 32-bit system bus accesses are supported.
[30:30]	sbaccess16	RO	1 when 16-bit system bus accesses are supported.
[31:31]	sbaccess8	RO	1 when 8-bit system bus accesses are supported.

sbaddress0 (System Bus Address0) Register

The **sbaddress0r** is a 32-bit register that contains the address used for memory access through system bus access. When the system bus manager is busy, writes to this register set sbbusyerror and have no other effect.

Fig. 9.1.11: *sbaddress0*

sbaddress1 (System Bus Address1) Register

The **sbaddress1** is a 32-bit register that contains the address used for memory access through system bus access. When the system bus manager is busy, writes to this register set sbbusyerror and have no other effect.

Fig. 9.1.12: *sbaddress1*

sbdata0 (System Bus Data0) Register

The **sbdata0** is a 32-bit register that holds data for system bus memory operations. It is used to write data during memory write operations and to read data during memory read operations. When the system bus manager is busy, accesses to this register set sbbusyerror and have no other effect.

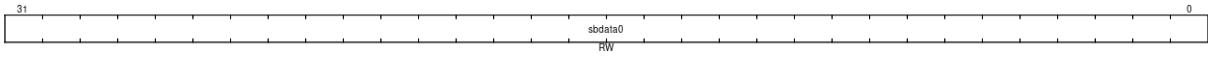


Fig. 9.1.13: *sbdata0*

sbdata1 (System Bus Data1) Register

The **sbdata1** is a 32-bit register that holds data for system bus memory operations. It is used to write data during memory write operations or to read data during memory read operations, enabling data transfer to or from the specified memory address during debugging.

Fig. 9.1.14: *sbdata1*

haltsum0 (Halt Summary 0) Register

The ** ** (Halt Summary 0) Register is a 32-bit read-only register where each bit indicates whether a specific hart is halted. Harts that are unavailable or nonexistent are not considered halted.

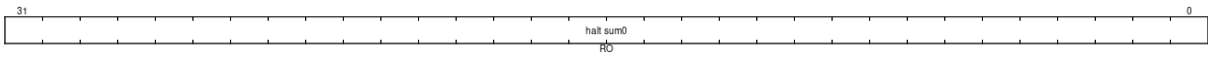


Fig. 9.1.15: *haltsum0*

9.1.3 Debugging Flow

1. **GDB** runs on the Debug Host and issues commands.
2. **OpenOCD** translates these commands and sends them over JTAG/SWD.
3. The **DTM** inside the target platform receives and processes these commands.
4. The **DM** halts/resumes harts, modifies registers, and executes debugging commands.
5. **Results** (register values, memory dumps, etc.) are sent back to GDB.

9.1.4 Troubleshooting

- **OpenOCD Fails to Connect** - Ensure the JTAG adapter is properly connected. - Use *dmesg* to check if the USB device is detected.
- **GDB Cannot Connect to OpenOCD** - Ensure OpenOCD is running before launching GDB. - Check the correct port (*localhost:3333*).

9.2 Itrace User Manual

Instruction Trace (Itrace) provides hardware support for tracing retired RISC-V instructions. The trace information can be filtered, compared, and stored in trace RAM for offline analysis. Itrace is primarily intended for debugging, profiling, and program flow analysis.

9.2.1 Instance Details

The table below lists the base addresses of the ITRACE core and RAM sink.

Table 9.2.1: ITRACE Instance Map

ITRACE Instance	Base Address
ITRACE	0x00060000
ITRACE_RAM	0x00060100

9.2.2 Register Map and Details

The table below lists the registers of the ITRACE along with their addresses and descriptions.

Table 9.2.2: ITRACE Register Map

Register Name	Offset	Length (In Bits)	Description
CTRL (Control Register)	0x00	16	Controls instruction trace operation. Enables trace generation, instruction tracing, and configures resynchronization behavior.
FILTER0_CTRL (Instruction Filter 0 Control Register)	0x08	32	Configures instruction trace filtering based on privilege level and comparator match conditions.
COMP1_CTRL (Comparator 1 Control Register)	0x0C	16	Configures Comparator 1 input sources, comparison modes, and notification behavior.
COMP2_CTRL (Comparator 2 Control Register)	0x10	16	Configures Comparator 2 input sources, comparison modes, and notification behavior.
COMP3_CTRL (Comparator 3 Control Register)	0x14	16	Configures Comparator 3 input sources, comparison modes, and notification behavior.

continues on next page

Table 9.2.2 – continued from previous page

Register Name	Offset	Length (In Bits)	Description
COMP1_PMATCH_LOW (Comparator 1 Primary Match Low Register)	0x18	32	Holds the lower 32 bits of the primary match value for Comparator 1.
COMP1_PMATCH_HIGH (Comparator 1 Primary Match High Register)	0x1C	32	Holds the upper 32 bits of the primary match value for Comparator 1.
COMP1_SMATCH_LOW (Comparator 1 Secondary Match Low Register)	0x20	32	Holds the lower 32 bits of the secondary match value for Comparator 1.
COMP1_SMATCH_HIGH (Comparator 1 Secondary Match High Register)	0x24	32	Holds the upper 32 bits of the secondary match value for Comparator 1.
COMP2_PMATCH_LOW (Comparator 2 Primary Match Low Register)	0x28	32	Holds the lower 32 bits of the primary match value for Comparator 2.
COMP2_PMATCH_HIGH (Comparator 2 Primary Match High Register)	0x2C	32	Holds the upper 32 bits of the primary match value for Comparator 2.
COMP2_SMATCH_LOW (Comparator 2 Secondary Match Low Register)	0x30	32	Holds the lower 32 bits of the secondary match value for Comparator 2.
COMP2_SMATCH_HIGH (Comparator 2 Secondary Match High Register)	0x34	32	Holds the upper 32 bits of the secondary match value for Comparator 2.

continues on next page

Table 9.2.2 – continued from previous page

Register Name	Offset	Length (In Bits)	Description
COMP3_PMATCH_LOW (Comparator 3 Primary Match Low Register)	0x38	32	Holds the lower 32 bits of the primary match value for Comparator 3.
COMP3_PMATCH_HIGH (Comparator 3 Primary Match High Register)	0x3C	32	Holds the upper 32 bits of the primary match value for Comparator 3.
COMP3_SMATCH_LOW (Comparator 3 Secondary Match Low Register)	0x40	32	Holds the lower 32 bits of the secondary match value for Comparator 3.
COMP3_SMATCH_HIGH (Comparator 3 Secondary Match High Register)	0x44	32	Holds the upper 32 bits of the secondary match value for Comparator 3.
COMP4_PMATCH_LOW (Comparator 4 Primary Match Low Register)	0x48	32	Holds the lower 32 bits of the primary match value for Comparator 4.
COMP4_PMATCH_HIGH (Comparator 4 Primary Match High Register)	0x4C	32	Holds the upper 32 bits of the primary match value for Comparator 4.
COMP4_SMATCH_LOW (Comparator 4 Secondary Match Low Register)	0x50	32	Holds the lower 32 bits of the secondary match value for Comparator 4.
COMP4_SMATCH_HIGH (Comparator 4 Secondary Match High Register)	0x54	32	Holds the upper 32 bits of the secondary match value for Comparator 4.

continues on next page

Table 9.2.2 – continued from previous page

Register Name	Offset	Length (In Bits)	Description
FILTER1_CTRL (Instruction Filter 1 Control Register)	0x58	32	Configures instruction trace filtering rules for Filter 1 using comparator match conditions.
FILTER2_CTRL (Instruction Filter 2 Control Register)	0x5C	32	Configures instruction trace filtering rules for Filter 2 using comparator match conditions.

The table below lists the registers of the ITRACE RAM along with their offsets and functional descriptions.

Table 9.2.3: ITRACE RAM Register Map

Register Name	Offset	Length (In Bits)	Description
CTRL (Control Register)	0x00	32	Controls ITRACE RAM operation. Enables RAM capture of trace packets and configures behavior when the RAM becomes full.
IMPL (Implementation Register)	0x04	32	Provides implementation and capability information such as version, component type, and supported memory features.
START_LOW (Start Address Low Register)	0x10	32	Holds the lower 32 bits of the trace RAM start address.
START_HIGH (Start Address High Register)	0x14	32	Holds the upper 32 bits of the trace RAM start address.
LIMIT_LOW (Limit Address Low Register)	0x18	32	Holds the lower 32 bits of the trace RAM end (limit) address.
LIMIT_HIGH (Limit Address High Register)	0x1C	32	Holds the upper 32 bits of the trace RAM end (limit) address.

continues on next page

Table 9.2.3 – continued from previous page

Register Name	Offset	Length (In Bits)	Description
WP_LOW (Write Pointer Low Register)	0x20	32	Holds the lower 32 bits of the current write pointer for trace packets stored in RAM.
WP_HIGH (Write Pointer High Register)	0x24	32	Holds the upper 32 bits of the current write pointer for trace packets stored in RAM.
RP_LOW (Read Pointer Low Register)	0x28	32	Holds the lower 32 bits of the current read pointer for trace packet retrieval.
RP_HIGH (Read Pointer High Register)	0x2C	32	Holds the upper 32 bits of the current read pointer for trace packet retrieval.
DATA (Trace Data Register)	0x40	32	Provides access to trace data stored in RAM for reading by an external host.
DMA_THRESH (DMA Threshold Register)	0x44	32	Configures the threshold level at which an interrupt is generated to trigger DMA transfer of trace data.

Control Register (CTRL)

The ITRACE_CTRL is a 16-bit register used to configure and control instruction trace operation. It enables trace generation, instruction tracing, and controls resynchronization behavior.

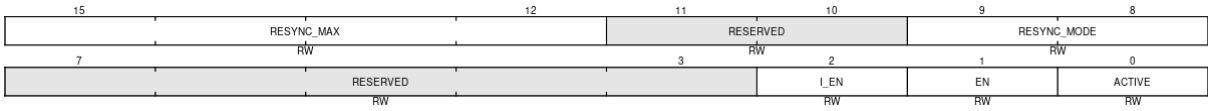


Fig. 9.2.1: CTRL

Table 9.2.4: ITRACE_CTRL Register Fields

Bit	Field Name	Permission	Description
[0:0]	ACTIVE	RW	Enable for the trace subsystem. When set to 1, the trace logic is active and accessible. When cleared to 0, the trace subsystem may be clock-gated or powered down, and access to other trace registers may be restricted.
[1:1]	EN	RW	Enables trace generation. When cleared, any pending trace data is flushed to the configured trace sink.
[2:2]	I_EN	RW	Enables instruction trace generation. Trace output is subject to any configured filtering conditions.
[9:8]	RESYNC_MODE	RW	Selects the resynchronization mechanism used to insert synchronization packets into the trace stream. • 0: Disabled • 1: Based on trace packet count • 2: Based on clock cycle count • 3: Based on retired instructions (16-bit half-words)
[15:12]	RESYNC_MAX	RW	Specifies the maximum interval between synchronization packets. The interval is calculated as: 2^RESYNC_MAX + 4 units, where the unit is defined by RESYNC_MODE.

Instruction Filter 0 Control Register (FILTER0_CTRL)

The FITER_CTRL register is a 32-bit register that configures instruction trace filtering. It selects comparator outputs, privilege matching, and enables filtering logic used to qualify trace generation.

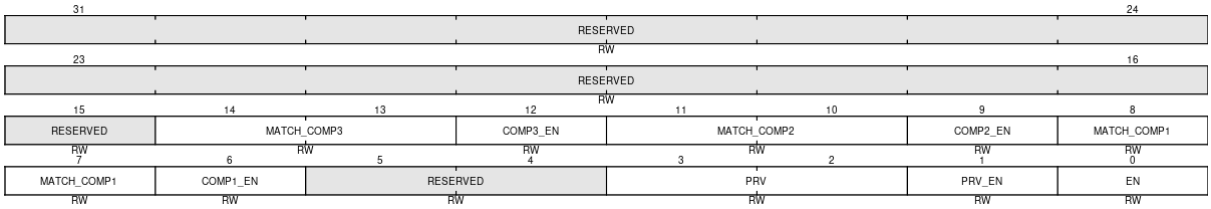


Fig. 9.2.2: FILTER0_CTRL

Table 9.2.5: FILTER0_CTRL Register Fields

Bits	Field Name	Permission	Description
[0:0]	EN	RW	Global enable for instruction trace filtering.
[1:1]	PRV_EN	RW	Enables privilege-level based filtering.
[3:2]	PRV	RW	Specifies the privilege level to be matched when PRV_EN is set.
[6:6]	COMP1_EN	RW	Enables comparator 1 for trace filtering.
[8:7]	MATCH_COMP	RW	Selects the comparator output used for the first filter condition.
[9:9]	COMP2_EN	RW	Enables comparator 2 for trace filtering.
[11:10]	MATCH_COMP	RW	Selects the comparator output used for the second filter condition.
[12:12]	COMP3_EN	RW	Enables comparator 3 for trace filtering.
[14:13]	MATCH_COMP	RW	Selects the comparator output used for the third filter condition.
[31:15]	RESERVED	RW	Reserved

Comparator 1 Control Register (COMP1_CTRL)

The COMP1_CTRL register is a 16-bit register that configures the behavior of Comparator 1. It selects the primary and secondary inputs, defines comparison functions, determines how comparator results are combined, and controls optional trace notification generation.

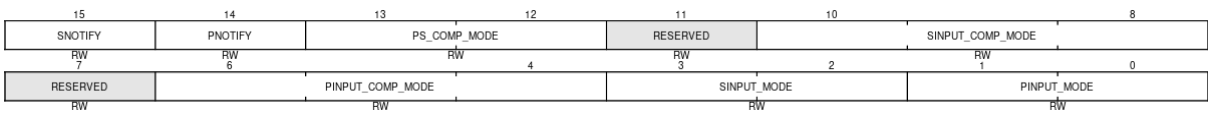


Fig. 9.2.3: COMP1_CTRL

Table 9.2.6: COMP1_CTRL Register Fields

Bits	Field Name	Permission	Description
[1:0]	PINPUT_MODE	RW	Selects the primary comparator input source. • 0: Instruction address • 1: Context • 2: Trap value • 3: Data address
[3:2]	SINPUT_MODE	RW	Selects the secondary comparator input source. • 0: Instruction address • 1: Context • 2: Trap value • 3: Data address
[6:4]	PINPUT_COMP_MODE	RW	Selects the comparison function applied to the primary input. • 0: Equal • 1: Not equal • 2: Less than • 3: Less than or equal • 4: Greater than • 5: Greater than or equal • 6: Always false (primary input ignored) • 7: Always true (primary input ignored)

continues on next page

Table 9.2.6 – continued from previous page

Bits	Field Name	Permission	Description
[10:8]	SINPUT_COMP_MODE	RW	Selects the comparison function applied to the secondary input. • 0: Equal • 1: Not equal • 2: Less than • 3: Less than or equal • 4: Greater than • 5: Greater than or equal • 6: Always false (secondary input ignored) • 7: Always true (secondary input ignored)
[13:12]	PS_COMP_MODE	RW	Selects how primary and secondary comparison results are combined. • 0: Primary result only (P) • 1: Primary AND secondary result (P && S) • 2: NOT (Primary AND secondary) (!(P && S)) • 3: Latch primary result until secondary result becomes true
[14:14]	PNOTIFY	RW	Generates a trace notification packet when the primary comparator matches (watchpoint).
[15:15]	SNOTIFY	RW	Generates a trace notification packet when the secondary comparator matches (watchpoint).

Comparator 2 Control Register (COMP2_CTRL)

The COMP2_CTRL register is a 16-bit register that configures the behavior of Comparator 2. It selects the primary and secondary inputs, defines comparison functions, determines how comparator results are combined, and controls optional trace notification generation.

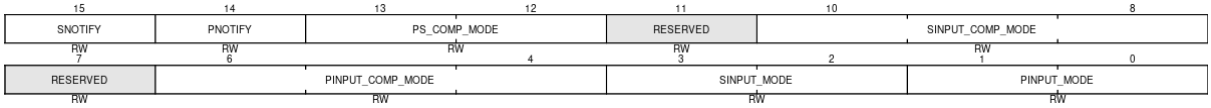


Fig. 9.2.4: COMP2_CTRL

Table 9.2.7: COMP1_CTRL Register Fields

Bits	Field Name	Permission	Description
[1:0]	PINPUT_MODE	RW	Selects the primary comparator input source. • 0: Instruction address • 1: Context • 2: Trap value • 3: Data address
[3:2]	SINPUT_MODE	RW	Selects the secondary comparator input source. • 0: Instruction address • 1: Context • 2: Trap value • 3: Data address

continues on next page

Table 9.2.7 – continued from previous page

Bits	Field Name	Permission	Description
[6:4]	PIN-PUT_COMP_MODE	RW	Selects the comparison function applied to the primary input. • 0: Equal • 1: Not equal • 2: Less than • 3: Less than or equal • 4: Greater than • 5: Greater than or equal • 6: Always false (primary input ignored) • 7: Always true (primary input ignored)
[10:8]	SIN-PUT_COMP_MODE	RW	Selects the comparison function applied to the secondary input. • 0: Equal • 1: Not equal • 2: Less than • 3: Less than or equal • 4: Greater than • 5: Greater than or equal • 6: Always false (secondary input ignored) • 7: Always true (secondary input ignored)

continues on next page

Table 9.2.7 – continued from previous page

Bits	Field Name	Permission	Description
[13:12]	PS_COMP_MODE	RW	Selects how primary and secondary comparison results are combined. • 0: Primary result only (P) • 1: Primary AND secondary result (P && S) • 2: NOT (Primary AND secondary) (!(P && S)) • 3: Latch primary result until secondary result becomes true
[14:14]	PNOTIFY	RW	Generates a trace notification packet when the primary comparator matches (watchpoint).
[15:15]	SNOTIFY	RW	Generates a trace notification packet when the secondary comparator matches (watchpoint).

Comparator 3 Control Register (COMP3_CTRL)

The COMP3_CTRL register is a 16-bit register that configures the behavior of Comparator 3. It selects the primary and secondary inputs, defines comparison functions, determines how comparator results are combined, and controls optional trace notification generation.

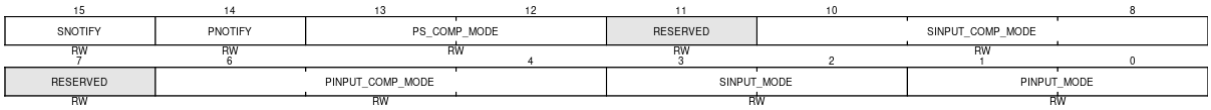


Fig. 9.2.5: COMP3_CTRL

Table 9.2.8: COMP3_CTRL Register Fields

Bits	Field Name	Permission	Description
[1:0]	PINPUT_MODE	RW	Selects the primary comparator input source. • 0: Instruction address • 1: Context • 2: Trap value • 3: Data address
[3:2]	SINPUT_MODE	RW	Selects the secondary comparator input source. • 0: Instruction address • 1: Context • 2: Trap value • 3: Data address
[6:4]	PINPUT_COMP_MODE	RW	Selects the comparison function applied to the primary input. • 0: Equal • 1: Not equal • 2: Less than • 3: Less than or equal • 4: Greater than • 5: Greater than or equal • 6: Always false (primary input ignored) • 7: Always true (primary input ignored)

continues on next page

Table 9.2.8 – continued from previous page

Bits	Field Name	Permission	Description
[10:8]	SINPUT_COMP_MODE	RW	Selects the comparison function applied to the secondary input. • 0: Equal • 1: Not equal • 2: Less than • 3: Less than or equal • 4: Greater than • 5: Greater than or equal • 6: Always false (secondary input ignored) • 7: Always true (secondary input ignored)
[13:12]	PS_COMP_MODE	RW	Selects how primary and secondary comparison results are combined. • 0: Primary result only (P) • 1: Primary AND secondary result (P && S) • 2: NOT (Primary AND secondary) (!(P && S)) • 3: Latch primary result until secondary result becomes true
[14:14]	PNOTIFY	RW	Generates a trace notification packet when the primary comparator matches (watchpoint).
[15:15]	SNOTIFY	RW	Generates a trace notification packet when the secondary comparator matches (watchpoint).

Comparator 1 Primary Match – Low Register (COMP1_PMATCH_LOW)

The COMP1_PMATCH_LOW register is a 32-bit register that stores the lower 32 bits of the primary match value for Comparator 1. This value is compared against the selected primary input (instruction address, data address, context, or trap value) based on the comparator configuration.

Fig. 9.2.6: COMP1_PMATCH_LOW

Comparator 1 Primary Match – High Register (COMP1_PMATCH_HIGH)

The COMP1_PMATCH_HIGH register is a 32-bit register that stores the upper 32 bits of the primary match value for Comparator 1. Together with COMP1_PMATCH_LOW, it forms a 64-bit primary comparison value.

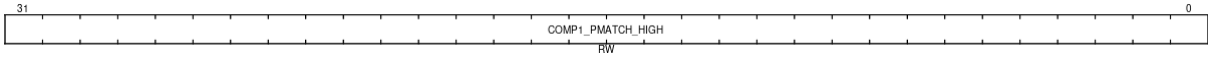


Fig. 9.2.7: COMP1_PMATCH_HIGH

Comparator 1 Secondary Match – Low Register (COMP1_SMATCH_LOW)

The COMP1_SMATCH_LOW register is a 32-bit register that stores the lower 32 bits of the secondary match value for Comparator 1. This value is used when secondary comparison logic is enabled and selected in the comparator control register.

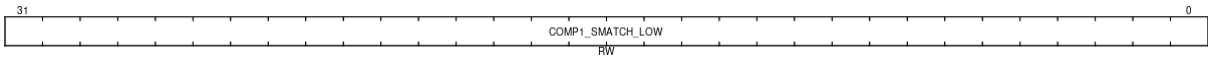


Fig. 9.2.8: COMP1_SMATCH_LOW

Comparator 1 Secondary Match – High Register (COMP1_SMATCH_HIGH)

The COMP1_SMATCH_HIGH register is a 32-bit register that stores the upper 32 bits of the secondary match value for Comparator 1. Together with COMP1_SMATCH_LOW, it forms a 64-bit secondary comparison value used by the secondary comparator logic.

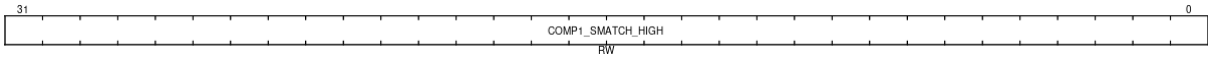


Fig. 9.2.9: COMP1_SMATCH_HIGH

Comparator 2 Primary Match – Low Register (COMP2_PMATCH_LOW)

The COMP2_PMATCH_LOW register is a 32-bit register that stores the lower 32 bits of the primary match value for Comparator 2. This value is compared against the selected primary input as configured in the Comparator 2 control register.

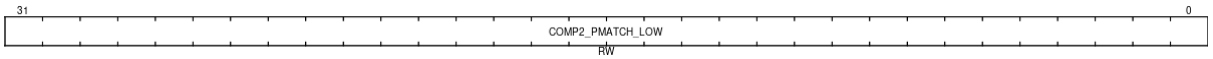


Fig. 9.2.10: COMP2_PMATCH_LOW

Comparator 2 Primary Match – High Register (COMP2_PMATCH_HIGH)

The COMP2_PMATCH_HIGH register is a 32-bit register that stores the upper 32 bits of the primary match value for Comparator 2. Together with COMP2_PMATCH_LOW, it forms a 64-bit primary comparison value.

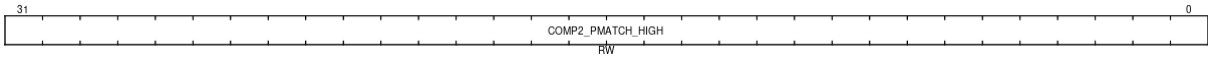


Fig. 9.2.11: COMP2_PMATCH_HIGH

Comparator 2 Secondary Match – Low Register (COMP2_SMATCH_LOW)

The COMP2_SMATCH_LOW register is a 32-bit register that stores the lower 32 bits of the secondary match value for Comparator 2. This value is used when secondary comparison logic is enabled for Comparator 2.

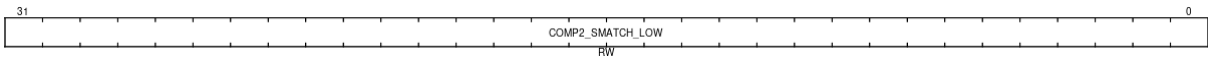


Fig. 9.2.12: COMP2_SMATCH_LOW

Comparator 2 Secondary Match – High Register (COMP2_SMATCH_HIGH)

The COMP2_SMATCH_HIGH register is a 32-bit register that stores the upper 32 bits of the secondary match value for Comparator 2. Together with COMP2_SMATCH_LOW, it

forms a 64-bit secondary comparison value.

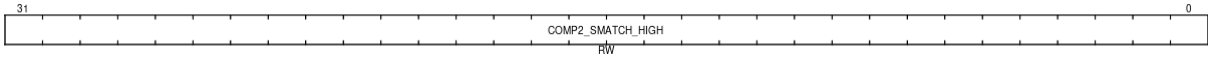


Fig. 9.2.13: COMP2_SMATCH_HIGH

Comparator 3 Primary Match – Low Register (COMP3_PMATCH_LOW)

The COMP3_PMATCH_LOW register is a 32-bit register that stores the lower 32 bits of the primary match value for Comparator 3. This value is compared against the selected primary input for Comparator 3.

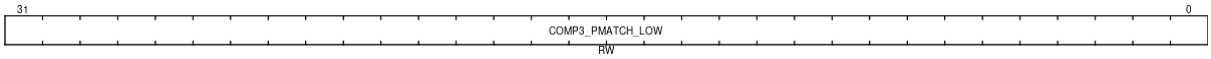


Fig. 9.2.14: COMP3_PMATCH_LOW

Comparator 3 Primary Match – High Register (COMP3_PMATCH_HIGH)

The COMP3_PMATCH_HIGH register is a 32-bit register that stores the upper 32 bits of the primary match value for Comparator 3. Together with COMP3_PMATCH_LOW, it forms a 64-bit primary comparison value.

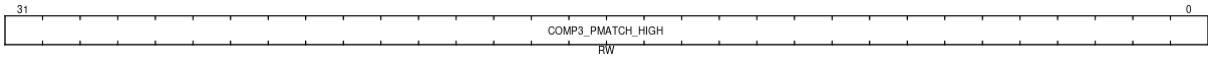


Fig. 9.2.15: COMP3_PMATCH_HIGH

Comparator 3 Secondary Match – Low Register (COMP3_SMATCH_LOW)

The COMP3_SMATCH_LOW register is a 32-bit register that stores the lower 32 bits of the secondary match value for Comparator 3. This register is used when secondary comparison logic is enabled for Comparator 3.

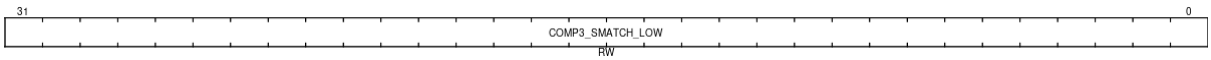


Fig. 9.2.16: COMP3_SMATCH_LOW

Comparator 3 Secondary Match – High Register (COMP3_SMATCH_HIGH)

The COMP3_SMATCH_HIGH register is a 32-bit register that stores the upper 32 bits of the secondary match value for Comparator 3. Together with COMP3_SMATCH_LOW, it forms a 64-bit secondary comparison value.

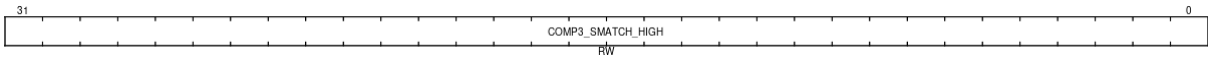


Fig. 9.2.17: COMP3_SMATCH_HIGH

Comparator 4 Primary Match – Low Register (COMP4_PMATCH_LOW)

The COMP4_PMATCH_LOW register is a 32-bit register that stores the lower 32 bits of the primary match value for Comparator 4. This value is compared against the selected primary input based on the Comparator 4 configuration.

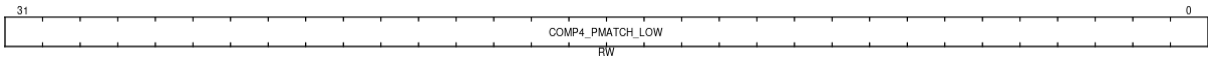


Fig. 9.2.18: COMP4_PMATCH_LOW

Comparator 4 Primary Match – High Register (COMP4_PMATCH_HIGH)

The COMP4_PMATCH_HIGH register is a 32-bit register that stores the upper 32 bits of the primary match value for Comparator 4. Together with COMP4_PMATCH_LOW, it forms a 64-bit primary comparison value.

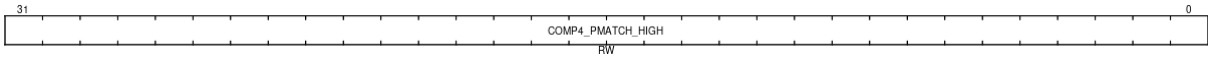


Fig. 9.2.19: COMP4_PMATCH_HIGH

Comparator 4 Secondary Match – Low Register (COMP4_SMATCH_LOW)

The COMP4_SMATCH_LOW register is a 32-bit register that stores the lower 32 bits of the secondary match value for Comparator 4. This value is used when secondary comparison logic is enabled for Comparator 4.

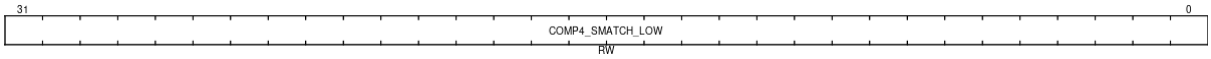


Fig. 9.2.20: COMP4_SMATCH_LOW

Comparator 4 Secondary Match – High Register (COMP4_SMATCH_HIGH)

The COMP4_SMATCH_HIGH register is a 32-bit register that stores the upper 32 bits of the secondary match value for Comparator 4. Together with COMP4_SMATCH_LOW, it forms a 64-bit secondary comparison value.

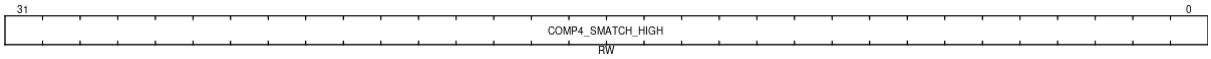


Fig. 9.2.21: COMP4_SMATCH_HIGH

Filter 1 Control Register (FILTER1_CTRL)

The FILTER1_CTRL register configures the first instruction trace filter. It selects comparator outputs, privilege matching, and enables filtering logic used to qualify trace generation.

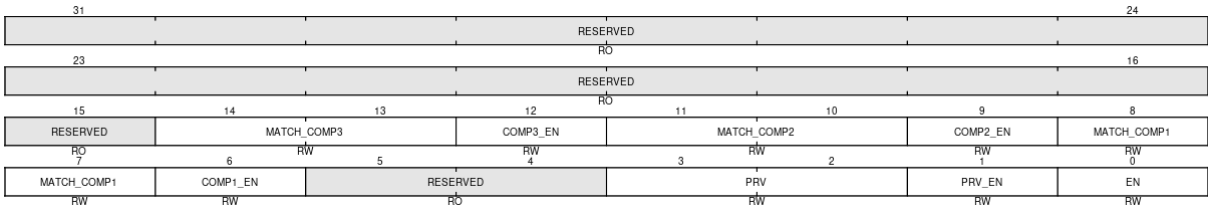


Fig. 9.2.22: FILTER1_CTRL

Table 9.2.9: FILTER1_CTRL Register Fields

Bits	Field Name	Permission	Description
[0:0]	EN	RW	Global enable for instruction trace filtering.
[1:1]	PRV_EN	RW	Enables privilege-level based filtering.
[3:2]	PRV	RW	Specifies the privilege level to be matched when PRV_EN is set.

continues on next page

Table 9.2.9 – continued from previous page

Bits	Field Name	Permission	Description
[6:6]	COMP1_EN	RW	Enables comparator 1 for trace filtering.
[8:7]	MATCH_COMP	RW	Selects the comparator output used for the first filter condition.
[9:9]	COMP2_EN	RW	Enables comparator 2 for trace filtering.
[11:10]	MATCH_COMP	RW	Selects the comparator output used for the second filter condition.
[12:12]	COMP3_EN	RW	Enables comparator 3 for trace filtering.
[14:13]	MATCH_COMP	RW	Selects the comparator output used for the third filter condition.
[31:15]	RESERVED	RW	Reserved

Filter 2 Control Register (FILTER2_CTRL)

The FILTER2_CTRL register configures the second instruction trace filter. It selects comparator outputs, privilege matching, and enables filtering logic used to qualify trace generation.

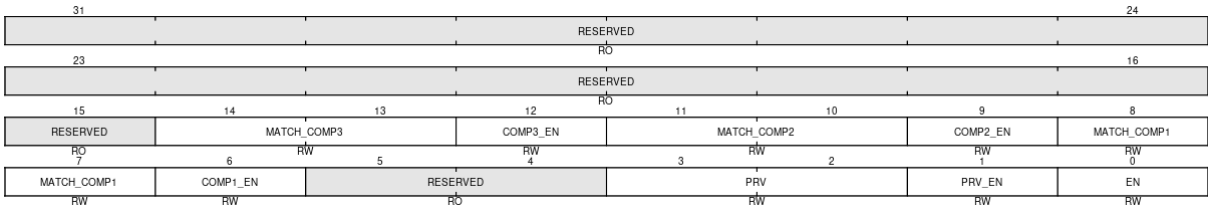


Fig. 9.2.23: FILTER2_CTRL

Table 9.2.10: FILTER2_CTRL Register Fields

Bits	Field Name	Permission	Description
[0:0]	EN	RW	Global enable for instruction trace filtering.

continues on next page

Table 9.2.10 – continued from previous page

Bits	Field Name	Permission	Description
[1:1]	PRV_EN	RW	Enables privilege-level based filtering.
[3:2]	PRV	RW	Specifies the privilege level to be matched when PRV_EN is set.
[6:6]	COMP1_EN	RW	Enables comparator 1 for trace filtering.
[8:7]	MATCH_COMP	RW	Selects the comparator output used for the first filter condition.
[9:9]	COMP2_EN	RW	Enables comparator 2 for trace filtering.
[11:10]	MATCH_COMP	RW	Selects the comparator output used for the second filter condition.
[12:12]	COMP3_EN	RW	Enables comparator 3 for trace filtering.
[14:13]	MATCH_COMP	RW	Selects the comparator output used for the third filter condition.
[31:15]	RESERVED	RW	Reserved

Control Register (CTRL)

The CTRL register is a 32-bit control register that manages the operational state of the ITRACE RAM. It enables trace packet capture into RAM and controls behavior when the RAM reaches its capacity.

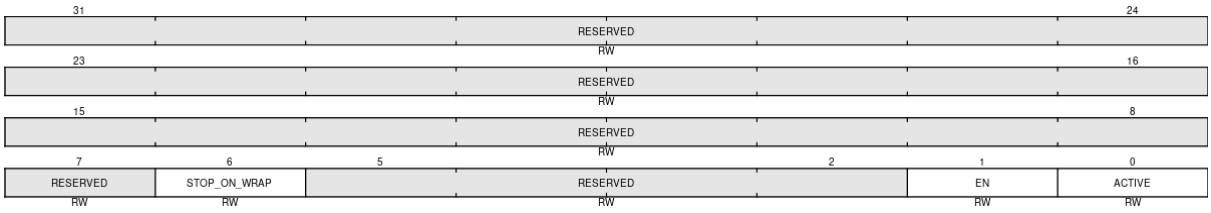


Fig. 9.2.24: CTRL

Table 9.2.11: CTRL Register Fields

Bits	Field Name	Permission	Description
[0:0]	ACTIVE	RW	Indicates whether the ITRACE RAM is active. When cleared, RAM operation may be gated or inactive.
[1:1]	EN	RW	Enables the RAM sink to accept and store trace packets.
[5:2]	RESERVED	RW	Reserved.
[6:6]	STOP_ON_WRAP	RW	When set, trace capture stops once the RAM becomes full. When cleared, trace data wraps around.
[31:7]	RESERVED	RW	Reserved.

Implementation Register (IMPL)

The IMPL register is a 32-bit, read-only register that reports implementation details and supported features of the ITRACE RAM sink.

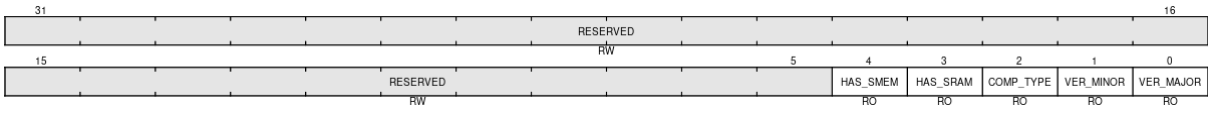


Fig. 9.2.25: IMPL

Table 9.2.12: IMPL Register Fields

Bits	Field Name	Permission	Description
[0:0]	VER_MAJOR	RO	Major version identifier of the ITRACE RAM implementation.
[1:1]	VER_MINOR	RO	Minor version identifier of the ITRACE RAM implementation.
[2:2]	COMP_TYPE	RO	Identifies the trace sink component type.

continues on next page

Table 9.2.12 – continued from previous page

Bits	Field Name	Permission	Description
[3:3]	HAS_SRAM	RO	Indicates whether internal SRAM storage is supported.
[4:4]	HAS_SMEM	RO	Indicates whether external or shared memory support is available.
[31:5]	RESERVED	RW	Reserved.

Start Address – Low Register (START_LOW)

The START_LOW register is a 32-bit register that holds the lower 32 bits of the start address of the trace RAM region. Together with START_HIGH, it defines the base address where trace packet storage begins.

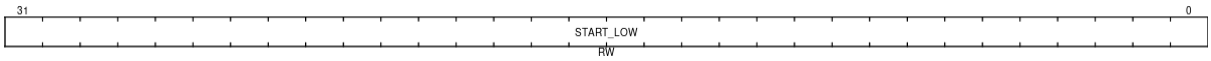


Fig. 9.2.26: START_LOW

Start Address – High Register (START_HIGH)

The START_HIGH register is a 32-bit register that holds the upper 32 bits of the trace RAM start address. This register is used in conjunction with START_LOW to form a complete 64-bit start address.

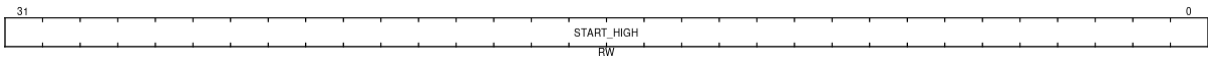


Fig. 9.2.27: START_HIGH

Limit Address Low Register (LIMIT_LOW)

The LIMIT_LOW register is a 32-bit register that stores the lower 32 bits of the trace RAM end address. It defines the lower portion of the memory boundary beyond which trace data must not be written.

Fig. 9.2.28: LIMIT_LOW

Limit Address – High Register (LIMIT_HIGH)

The LIMIT_HIGH register is a 32-bit register that holds the upper 32 bits of the trace RAM end address. Combined with LIMIT_LOW, it specifies the full 64-bit upper limit of the trace RAM region.

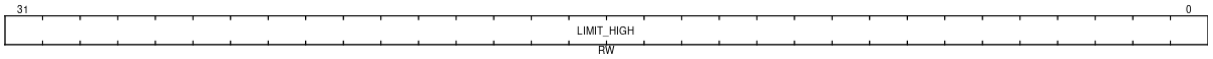


Fig. 9.2.29: LIMIT_HIGH

Write Pointer – Low Register (WP_LOW)

The WP_LOW register is a 32-bit register that contains the lower 32 bits of the current write pointer. It indicates where the next trace packet will be written in RAM.

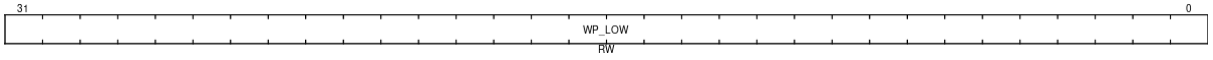


Fig. 9.2.30: WP_LOW

Write Pointer – High Register (WP_HIGH)

The WP_HIGH register is a 32-bit register that contains the upper 32 bits of the write pointer address. Together with WP_LOW, it forms the complete 64-bit write pointer.

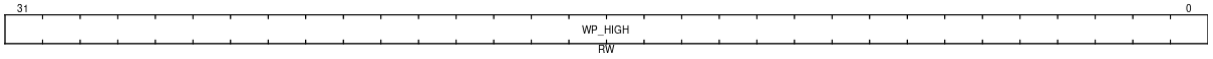


Fig. 9.2.31: WP_HIGH

Read Pointer – Low Register (RP_LOW)

The RP_LOW register is a 32-bit register that stores the lower 32 bits of the read pointer. It indicates the location from which trace data will be read by the host or DMA.

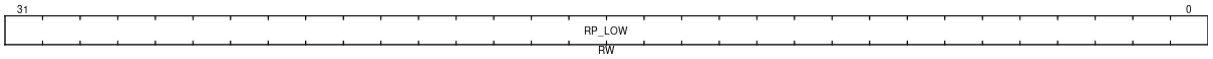


Fig. 9.2.32: RP_LOW

Read Pointer – High Register (RP_HIGH)

The RP_HIGH register is a 32-bit register that contains the upper 32 bits of the read pointer address. Combined with RP_LOW, it forms the full 64-bit read pointer for trace data retrieval.

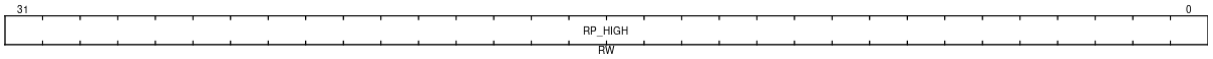


Fig. 9.2.33: RP_HIGH

Trace Data Register (DATA)

The DATA register is a 32-bit, read-only register used by an external host to retrieve trace data stored in RAM.

Fig. 9.2.34: DATA

DMA Threshold Register (DMA_THRESH)

The DMA_THRESH register is a 32-bit configuration register that defines the threshold at which the ITRACE RAM triggers a DMA interrupt to offload trace data.

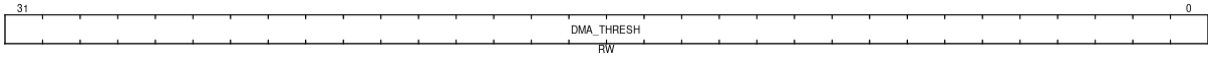


Fig. 9.2.35: DMA_THRESH

Chapter 10. Contributing

10.1 Contribution

1. Fork the Repository.
2. Commit Changes.
3. Put the description of changes made in CHANGELOG.md
4. Create Pull Requests
5. You have successfully Contributed to the Project.

10.2 Issues

If any Issues found, Create an issue in git repository given below, and get them solved...
Open issues here

Chapter 11. CHANGELOG

11.1 CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

11.1.1 [1.2.0] - 2026-02-19

Added

- NIL

Fixed

- NIL

Changed

- Updated peripheral template with doxygen syntax.
- Updated README.md with doxygen setup steps.

Removed

- NIL

11.1.2 [1.1.0] - 2025-01-11

Added

- Added the doc for OTP, CLINT, AES, RSA, SHA, and QSPI.

Fixed

- PDF Generation

- Waveform Generation in HTML

Changed

- C-Code in the documentations.

Removed

- NIL

11.1.3 [1.0.0] - 2024-11-13

Added

- Added new rst files for peripherals.

Fixed

- NIL

Changed

- NIL

Removed

- NIL

Chapter 12. Relevant links

The processor design follows the [1] and [2] specifications and also uses [3] for implementation ideas.

Chapter 13. Revision History

Table 13.1: Revision History

Version	Description	Reviewer	Sign off	Date
1.0	Initial version	M. Kapil Shyam	T.R. Shashwath	26-02-2025

BIBLIOGRAPHY

[1] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanović. The RISC-V instruction set manual, volume i: user-level ISA, version 2.1. Technical Report, RISC-V Foundation, 2016. URL: https://docs.riscv.org/reference/isa/v2.1/_attachments/riscv-unprivileged.pdf.

[2] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanović. The RISC-V instruction set manual, volume ii: privileged architecture, version 1.11. Technical Report, RISC-V Foundation, 2019. URL: https://docs.riscv.org/reference/isa/v1.11/_attachments/riscv-privileged.pdf.

[3] Shakti Development Team. The Shakti processor project: open-source RISC-V c-class core. 2021. An open-source processor design and research initiative by IIT Madras. URL: https://gitlab.com/shaktiproject/cores/c-class.git.