**NuPRISM Xilinx Firmware Architecture**

**Author(s):** Ryan Payne

**Release:** 1.4

**Release Date:** 19/05/2020

**Keywords:** IWCD, NuPRISM, Firmware, Xilinx, Altera, Ultrasclae+, Zynq, MAX10, Main Board, Electronics, mPMT

**History of Changes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Release No.** | **Date** | **Description** | **Authour(s)** |
| 1.1 | 15/05/2020 | Initial Release | Ryan Payne |
| 1.2 | 19/05/2020 | More detailed descriptions of components | Ryan Payne |
| 1.3 | 19/05/2020 | Expanded on networking  Updating figure 1-3  Adding details about the clock distribution | Ryan Payne |
| 1.4 | 19/05/2020 | Updated preface and introduction with more accurate information | Ryan Payne |

# Preface

This document is meant to act as an overview of the Xilinx-based NuPRISM mPMT Main Board firmware. An introduction to the experiment, purpose of the electronics, and main components is followed by a concept of operations for the main board electronics. An architecture diagram is provided, and each component is described in further detail. This document is not intended to act as the definitive architecture, but rather a living document to be revised upon additional discussion, consultation, and experimentation.

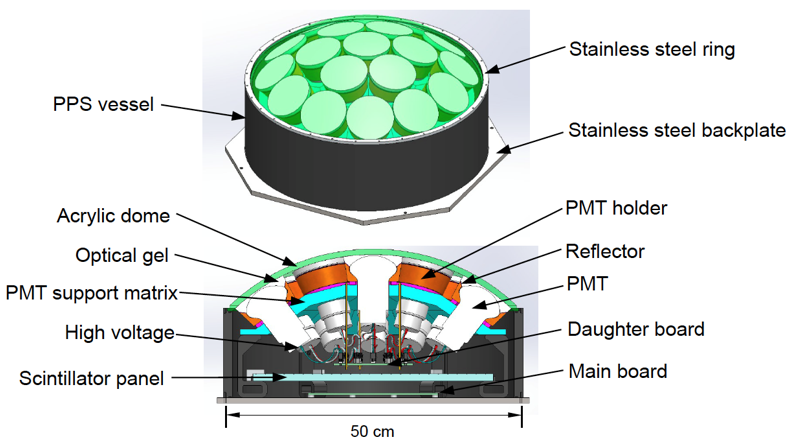
# Introduction

The IWCD experiment will place a water Cherenkov detector in a deep vertical pit in Japan. Approximately 500 multi-photomultiplier tubes (mPMT) make up the Cherenkov detectors photo-sensors. The experiment setup is shown in figure 1-1. Each mPMT consists of 20 photomultiplier tubes connected to the main board electronics. The mPMT is shown in figure 1-2. A mPMT concentrator card is responsible for providing higher level control and monitoring of the mPMTs. The experiment will be observing controlled events generated by a neutrino beam and a LED-based calibration source. It will also observe more random events from cosmic-ray muons.

A picture containing table

Description automatically generated

**Figure 1-1:** IWCD experiment setup



**Figure 1-2:** Diagram of the mPMT

From an electronics development perspective, the main board is a data acquisition system, located in a difficult to reach place, and needs to consume as little power as necessary given that it is part of a much larger system. Therefore, the firmware needs to be:

* Robust
* Reprogrammable
* Low power

A high-level hardware block diagram is shown in figure 1-3. The SoC deserializes and processes data from 5 quad-channel ADCs. The ADC clock is generated by a clock-cleaner which can derive its clock from a crystal oscillator, SMA input, RJ45 input, or POE input. For connectivity, the SoC provides connection to RJ45, USB, Ethernet, and SFP. For system monitoring, the SoC is connected by I2C to several temperature, humidity, pressure and voltage sensors. The MAX10 provides the SoC with UART connection to each PMT base, and has additional connection to the I2C bus, clock cleaner, and system clocks.

A screenshot of a cell phone

Description automatically generated

**Figure 1-3:** High-level hardware block diagram for mPMT main board

The clock cleaner takes its inputs from the SMA clock, RJ45 clock, POE clock, and a reference oscillator. The SMA clock and RJ45 clock are muxed and selected by the MAX10. On reset/startup the clock cleaner will use the oscillator as a reference. If either of the other two inputs are detected as valid, it will switch its reference to the valid clock. If both inputs are valid, a user-settable priority bit is referenced. While the input and output clocks are locked, device GPIO is activated and can be monitored. The generated clock is distributed to the 5 ADCs and used as a sample clock. The clock is also distributed to the MAX10 where it can be monitored or used for other functionality.

The main processor currently in place is the Xilinx Zynq Ultrascale+ EG MPSoC. A block diagram depicting its internals is shown in figure 1-4. The MPSoC consists of a processing system (PS) and programmable logic (PL). The PL is the FPGA portion of the MPSoC and contains RAM blocks, DSP blocks, and high-speed connectivity blocks. The PS is a highly integrated system. Most notable it contains it a quad-core 64-bit application processing unit (APU), and dual-core 32-bit real-time processing unit (RPU). To manage the power consumption of this device, the platform management unit (PMU) can be used to suspend, wakeup, and power-down different components of the entire architecture. There are several ways to manage a wakeup of the APU. Some notable methods are by GPIO input or by request from the RPU.

A screenshot of a cell phone

Description automatically generated

**Figure 1-4:** Zynq Ultrascale+ EG MPSoC architecture diagram

# Concept of Operations

The concept of operations is made up of three different modes of operation. Idle mode, high frequency mode (HF) mode, and low frequency (LF) mode.

## Idle Mode

The idle mode flow chart is shown in figure 2-1. Idle mode is expected to be used when the experiment is undergoing maintenance, setup, or any other scenario where experimental data is not being collected while the system has power. It is expected that no PMT data will be collected at this time, but general system health is being monitored.

In idle mode, the ADCs and clock cleaner are powered down to save power. The APU is typically suspended. Periodically, the RPU will wake up the APU to collect system health data and deliver that data over the network to a higher-level control and monitoring system. By having the APU run only when necessary, the RMS power consumed by the electronics is reduced.

## High Frequency Mode

High frequency mode is expected to be used in scenarios where the system expects to see a high event rate in some or all mPMTs. Typically, this mode would be used when the PMTs are being calibrated by a LED source or when the experiment is running under a beam.

In high frequency mode, the ADCs and clock cleaner are running. The PL is running the DSP algorithms. The APU is typically processing data on an external or DSP generated trigger. The APU may be suspended if there have been no external or internal triggers for a specified period.

If the APU is suspended and a trigger occurs, the PL writes the data to the APU memory and wakes up the APU. The APU then compresses and packetizes that data and delivers it over the network. The APU will now have to wait the specified period before it is suspended. Periodically, he RPU will wake up the APU to collect system health data and deliver that data over the network to a higher-level control and monitoring system.

This mode of operation allows the system to process high event rates more readily. Given that Cherenkov light will not produce events in all mPMTs, the APU suspension allows those inactive mPMTs to limit their power consumption. The method in which this is implemented, with a required no-event-duration, gives the mPMT hysteresis. Here, the system recognizes that if an event occurred, another is likely to follow.

## Low Frequency Mode

Low frequency mode is expected to be used in scenarios where the system expects to see more random events which are triggered at a lower event rate. It is expected that this mode would be used when there is no beam or calibration, and cosmic rays are being observed.

This mode would be similar to the high frequency mode with no, or minimal, hysteresis.

# Firmware Architecture

A screenshot of a cell phone

Description automatically generated

**Figure 3-1:** Firmware Architecture Diagram

## 3.1 APU

The APU will be running Linux to simplify network connectivity and the modularization of the firmware. The Linux image will be built using Petalinux, Xilinx’s Linux build environment. Petalinux provides device specific support for the Zynq Ultrascale+ MPSoC. The APU is part of the full-power domain in the Ultrascale+ and should only be ran when necessary to minimize power consumption. The APU is responsible for ensuring it is ready to be suspended, determining a wakeup source, and finally suspending itself. The APU should be powered on periodically to check system health, and non-periodically when it needs to either respond to an event, control the system, or update the firmware.

### 3.1.1 APU – Drivers

Linux device drivers will need to be written to interact with the following devices:

* PMTs (UART + Select Signal)
* MAX10 (I2C or SPI)
* ADCs (SPI)
* Clock Cleaner (I2C)
* Voltage Sensors (I2C)
* Temperature Sensors (I2C)
* Humidity Sensors (I2C)
* Pressure Sensor(I2C)

Less generic drivers will need to be written to interact with:

* Deserializer (AXI)
* DSP
* Various Triggers

### 3.1.2 APU – Main Application

The main application will be responsible determining when it should self-suspend. This logic will change depending on the mode it is running in (see Section 2). The program should wake up to respond to events, monitor system health, and respond to higher level control signals.

The event-response-wakeup signal will be generated by the DSP or by the higher-level control signals. When this occurs, the APU will wake up, compress and packetize the data, and transmit the data over the network. After this, it may self-suspend or stay awake for a specified period of time if it is expecting more events to occur.

Regardless of if events are occurring, the system should periodically wakeup to monitor system health in the form of temperate, pressure, humidity, and voltage sensors. It should also ensure all control values (PMT settings, deserializer delay, DSP settings) are as expected. This data should be transmitted over the network to allow operators to monitor the system.

Higher level control signals should act to wake up the system, which should then respond accordingly. This could include:

* Updating control values
* Terminating the main application
* Completing an unscheduled system health check

The data packets will be sent in UDP form and the control signals will use TCP.

### 3.1.3 APU – Firmware Updater

The APU should be able to support the remote updating of the firmware. This will include reprogramming the PL, updating the application software, and programming a new Linux image.

## 3.2 RPU

The RPU will be running a standalone C/C++ program which will be responsible for waking up the APU to respond to events and check system health.

The RPU may also act as a watchdog for the APU. If the RPU does not have its APU watchdog requirements met, it will reset the APU.

To add robustness to the system and further limit the power consumption, the RPU may also collect system health itself to ensure sensor values are as expected when the APU is not running. If this were the case, the time between APU system health checks would increase.

## 3.3 MAX10

The MAX10 will be running very simple logic to mainly support the SoC. Its main operations will be to act as a mux for the SoC to PMT UART communications and provide various connections for the SoC. Given that it also has connections to many of the system clocks, it should be able to monitor the frequency of those clocks and generate valuable system health data.

### 3.3.1 MAX10 – UART Mux

The UART Mux should provide UART connection from the SoC to one of the 20 PMT bases. The select signal for this mux should be UART\_SEL[4:0].

### 3.3.2 MAX10 – SoC Support

The MAX10 provides connection to the SMA inputs and clock cleaner. These should be relayed to the SoC for more complex operations.

### 3.3.3 MAX10 – Clock Frequency Monitor

The MAX10 has connections to each ADC clock, and the clock cleaner SMA input and output. By implementing a digital frequency monitor, the MAX10 can provide data useful for monitoring and debugging operations. The MAX10 can act as a I2C or SPI slave which stores this data for the SoC to collect.

## 3.4 PL

The PL should mainly deal with the DSP. It should deserialize the data arriving from the 5 ADCs. It should pass that data to a DSP platform which should perform DSP algorithms and either respond to or produce or respond to a trigger. On this trigger, it should write the data into the memory of the APU. This trigger may also be used to wake up the APU to perform additional operations on the data and send it over the network.

The PL should also provide connectivity for the PS in the form of GPIO, SPI, and I2C connections.

### 3.4.1 PL – Deserializer

The deserializer needs to deserialize 2-wire 1.8V LVDS DDR signals arriving at 750 Mbps. The deserialize should provide the following functionality for control and debugging:

* Reset signal
* Locked signal
* Delay control interface

The locked signal is activated when the deserializer has correlated the data streams with their frame clocks. The delay control interface will likely use an AXI interface to provide the APU the capability to tune the deserializer delay values. These values account for the routing delay between ADC signals and provide optimal signal integrity.

### 3.4.2 PL – Digital Signal Processing (DSP)

The DSP is responsible for running DSP algorithm and determining when to set the internal trigger. A full picture of the control and debugging signals required has not been formed at this time.

### 3.4.3 PL – Direct Memory Access (DMA)

The DMA allows data to be written into the memory of the PS to be processed and transmitted over the network. Xilinx provides support for the implementation of the DMA.

### 3.4.4 PL – Wakeup Logic

While the event trigger could act as an APU wake up, more specific wakeup logic could be added to the system. One possible option is allowing a certain amount of event data be collected before transmitting it over the network. This would reduce the systems power consumption.

### 3.4.5 PL – Miscilanious

The PL should also provide connectivity for the PS in the form of GPIO, SPI, and I2C connections. If useful it may also complete hardware acceleration tasks apart from DSP algorithms.