2020

# Implementation of a Hierarchical, Embedded, Cyber Attack Detection System for SPI Devices on Unmanned Aerial Systems

Jeremy J. Price

# IMPLEMENTATION OF A HIERARCHICAL, EMBEDDED, CYBER ATTACK DETECTION SYSTEM FOR SPI DEVICES ON-BOARD UNMANNED AERIAL SYSTEMS

By

JEREMY PRICE

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

VIRGINIA COMMONWEALTH UNIVERSITY
Department of Electrical and Computer Engineering

MAY 2020

## ACKNOWLEDGMENT

I would like to thank my advisor, Robert H. Klenke, PhD, for all of his help and guidance through this process. I have learned so much over the course of this project and I appreciate the opportunity to work on a system at this scale.

I would also like to thank Peter Truslow and Andy Fabian for always answering my endless supply of questions about the Aries hardware and software. I truly would not have been able to complete all that I set out to do with this project without their tireless support. Dr. Matthew L. Leccadito also provided me with invaluable support and guidance about HECAD and how best to implement the SPI components.

Finally, would like to thank my council, Dr. Carl R Elks and Dr. Ruixin Niu. Their guidance and advice helped this system and thesis be the best it could.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

IMPLEMENTATION OF A HIERARCHICAL, EMBEDDED, CYBER ATTACK DETECTION SYSTEM FOR SPI DEVICES ON-BOARD UNMANNED AERIAL SYSTEMS

Abstract

by Jeremy Price, B.S.
Virginia Commonwealth University
May 2020

Ph.D: Robert H. Klenke

Unmanned Aerial Systems (UAS) create security concerns as their roles expand in commercial, military, and consumer spaces. The need to secure these systems is recognized in the architecture for a Hierarchical, Embedded, Cyber Attack Detection (HECAD) system. HECAD passively monitors the communication between a flight controller and all its peripherals like sensors and actuators. It ensures the functionality of a UAS is within the set of defined behavior and reports all potential problems, whether the errors were caused by cyber attacks or other physical faults. A portion of the design for Serial Peripheral Interface (SPI) devices on board a flight control system is developed on an FPGA device. A wide range of cyber attacks and other faults are checked in SPI HECAD, implemented with VHDL and verified through use of the Integrated Logic Analyzer tool.

# Chapter One

# Introduction

Unmanned Aerial Systems (UAS) have grown in popularity as tools across many sectors. Having previously only existed in the military space, the security risks posed by UAS were not as pronounced to the general public. Now, with the widespread adoption in both commercial and consumer spaces, the vulnerabilities presented by these systems have needed to be addressed. Part of the concern is that UASs are safety-critical systems; they can potentially cause serious harm to people, wildlife, and the environment if either operated incorrectly or maliciously attacked. UASs are cyber-physical systems (CPSs), which is partly the reason they need to be safety-critical systems. CPSs combine electronic and computer components, usually digital micro-controllers, with sensors and actuators. The actuators in the case of UASs are typical rotors, which by themselves create safety issues but also compounds with how fast the UAS payload is moved by them. As a result of the increasing use of these systems in consumer space, the Federal Aviation Administration (FAA) has placed restrictions on who can fly UASs and how they can be used. This does not tackle some of the other core issues revolving around UAS safety and security. They are not the same level of regulations on Commercial-Off-the-Shelf (COTS) UAS products as there are on cars or planes, for example. Certainly there are even less for UASs built from individual components.

While the software components of CPSs certainly are a problem, the cyber portion deserves just as much scrutiny, if not more, when it comes to safety issues. It has been well

documented over the history of the growth and proliferation of computing systems that the software domain has infinitely more room for complexity. Complexity tends to invite disasters like the Ariane V explosion and other famous software failures. UASs especially can suffer from this problem as Flight Control Systems (FCS) grow more complex with additions like autopilot and other autonomous functions. Although restrictions put in place by the FAA for line-of-sight only flight mitigate issues such as an operator being able to take over a malfunctioning autopilot system, concerns still exist. It is particularly problematic when an autopilot has complete control over a system, which is generally required for fully autonomous functionality, allowing an attacker access to that same level of unfettered control. Back door entry and other means of this access can be inserted in many levels of the UAS software in many ways for COTS products and even custom built systems. Flight control code hosted on the cloud can be spoofed using a copy of the website containing malicious modifications.

The software domain is only part of the cyber components in a CPS. The individual sensors and other hardware are all surfaces upon which faults and attacks can occur. For a UAS, this can easily include sensors for airspeed, barometric pressure, GPS, acceleration, orientations, and temperature, with each having its own separate security concerns. There is also communication between the UAS and a ground control station which is vulnerable to being intercepted like in man-in-the-middle and replay attacks. GPS alone can suffer from many different documented cyber attacks like jamming, spoofing, and walk off attacks. One specific attack surface that is shared by all the components is their supply chain. This type of vulnerability comes from how manufactured components have a long chain of production, all the way from the product design to construction or fabrications. Malicious actors can slip in at any point along this supply chain and insert components or firmware that can give access, cause failures, or create atypical operation. These types of attacks are particularly hard to detect and protect against, as consumers have very little access to the intellectual property (IP) controlled hardware to validate that it has not been maliciously altered in

any way. This is of course true for COTS UAS, like fully built DJI drones, but even for custom built UAS, the individual components still have programmable integrated circuits and can be victims of supply chain attacks. As individual components, each one has its own set of problems and vulnerabilities; when integrated together into a system, these issues are only compounded. System integration has also presented its own set of problems, usually emergent behavior that can cause faults or create new attack surfaces.

Virginia Commonwealth University's (VCU) Unmanned Aerial Vehicle (UAV) Laboratory has developed a state-of-the-art flight control system named Aries Tiny FCS. Aries encompasses both the flight control hardware and code. It has undergone several revisions and the hardware now has the flight control processor and its sensor suite on-board a PCB. It features a fully functioning autopilot system for both fixed-wing flight and quadcopters. With its development, the abundant safety and security concerns associated with unmanned systems were considered in a 2017 PhD dissertation written by Dr. Leccadito [1]. In this dissertation the doctoral candidate proposed the architecture for a Hierarchical, Embedded, Cyber Attack Detection System (HECAD). HECAD is comprised of four different modules. Hardware Resource Integrity Monitor (HRIM) and the sensor specific Information Integrity Monitor (I2M) modules are are the lowest level of hierarchy in the HECAD system, which takes a bottom-up approach to detecting cyber attacks. The HRIM and I2M often will not be able to detect coordinated and sophisticated cyber attacks alone, but the top module of the hierarchy, the Functional Integrity Monitor (FIM), can by using the information passed up from the bottom levels. The FIM can differentiate between cyber attacks and natural faults by using this already verified information.

**Contributions of this Thesis**

The objective of this thesis is to develop, implement, and verify the HECAD modules responsible for monitoring the Serial Peripheral Interface (SPI) sensors on-board the Aries Tiny FCS. It outlines the types of faults and attacks that can occur at the different levels of

hierarchy along the sensor datapath to and from the FCS. Additionally, SPI HECAD aims to have the bus specific HRIM and I2M modules sufficiently implemented to be integrated into the entire HECAD system. Both of these components are implemented in a Hardware Description Language, VHDL, to be realized in an Field Programmable Gate Array (FPGA) device. The validation of the design is presented with the use of Xilinx's Integrated Logic Analyser (ILA) waveforms along with discussion of its function. Fault injection is performed with a custom written hardware model tailored to perform a variety of attacks. The results of the fault injection are compared against the correct operation of an SPI transaction and show the respective HECAD modules detecting the attacks. Auxiliary development on integration of the other sensors' HECAD components into a cohesive and documentation for the future use of the SPI components are performed, though not necessarily presented in this thesis.

# Chapter Two

# Background/Related Work

This thesis will use the traditional definitions of faults, errors, and failures in the context of dependable computing. Faults refer to the defect in the system. The source of faults varies, including design problems, natural deterioration, or intentional attacks. The symptoms of a fault and how it alters the behavior of a system is defined as the error. Errors are the detectable portion of chain inside the system boundary, so the lower levels of HECAD will be concerned with identifying these symptoms. Failures are the propagation of the fault and corresponding errors to the edge of the system boundary. For example, a fault in a PCB design can be an open circuit on one of the signals in an internal communication bus. This fault will likely cause errors to happen inside the system as either the data or control mechanisms for that bus will be disrupted. The failure is recognized to the outside world when the disrupted communication causes the system to behave in an incorrect way.

## 2.1 UAV Vulnerabilities

The taxonomy of cyber attacks on CPSs have been investigated in a number of ways. The use of Data Flow Diagrams was identified [2] as a good way of creating this taxonomy. A good portion of the work done on examining UAV vulnerabilities has focused on the networked components of UASs [3]–[7]. It is logical to be concerned about the communication

systems on the UAS, as the researchers have found. These are large sources of attack surfaces. Links between the Ground Control Station (GCS) and the FCS are susceptible to the typical host of attacks possible on wireless networks. Denial of Service, dropping packets, injections [8], fuzzing, and Man-in-the-Middle [9] to name a few, are all pressing security concerns for communication systems. While these are the predominant vulnerabilities to access total control of the aircraft, lower level attack surfaces still deserve consideration. The contributions of this thesis lie within this domain, closer to the sensor level.

Less work has been done involving vulnerabilities on the sensor level for specifically UASs. Of this work, GPS has received the most attention which is reasonable given the critical role that the sensor plays in UAS navigation. It still falls within the category of wireless communication and is vulnerable to similar problems. GPS specifically involves satellite communication which differs from FCS/GCS communication protocols and thus has a different set of vulnerabilities. Spoofing [10]–[12] is performed by matching GPS signals to the receiver and slowly taking over the communication link until the point that the attacker can send incorrect coordinates to the UAS for whatever kind of malicious purpose. GPS jamming [13]–[15], another focus of GPS vulnerabilities, involves taking down the communication altogether.

Less focus has been spent for other specific sensors' vulnerabilities on a UAS. However, a UAS shares many common security concerns with generic CPSs [16]–[18]. Research in that domain can be applied to UASs with some restrictions. The nature of autonomy creates additional security concerns that only a subset of CPSs need to consider. This research can be applied when considering the attack surfaces, it is only the detection and mitigation strategies that differ between autonomous and human controlled systems. These attack surfaces can reside in firmware or embedded in the hardware of the sensors. So little research can be performed on documenting these vulnerabilities due to the size of the set containing all possible manifestations of these attacks. Supply Chain attacks [19], [20] can affect any and all aspects of a design which makes them as difficult to catalogue as they are to detect.

Other monitoring architectures [21]. Kwon et al. looks at several different scenarios of

potential cyber attack methods. Its work is of note because it deals with stealthy attacks, where the attacker attempts to mask the attack until a more complete failure is achieved, typically by fuzzing sensor information until the UAV's state estimation fails. The most effective types of attacks need to be coordinated across multiple sensors, requiring hardware access. Kwon et al. identify that the best way to defend against against these attacks would be more robust communication channels and redundancy in networked components, all of which the entire HECAD design aims to do.

## 2.2 HECAD Architecture

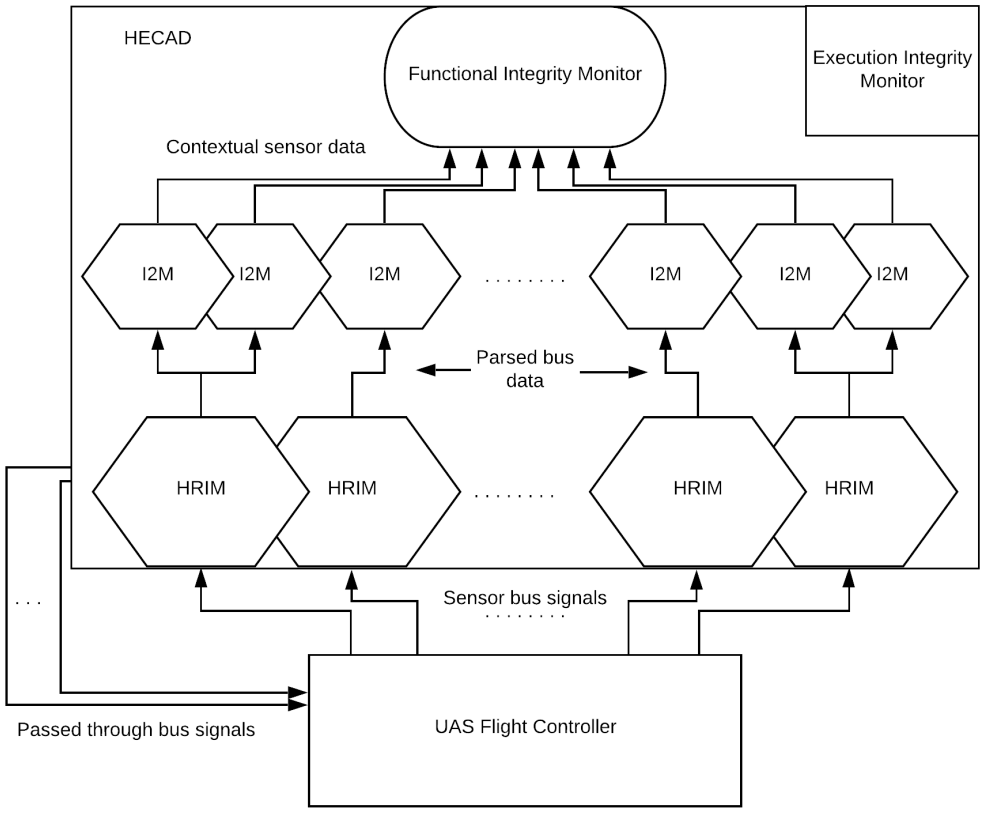


**Figure 2.1** Block Diagram of HECAD Datapath

Dr. Leccadito's 2017 PhD dissertation [1], titled "A Hierarchical Architectural Framework for Securing Unmanned Aerial Systems," provides the basis for HECAD. This system is designed to passively monitor all of the communication going on around a UAS's flight controller. It takes in this information straight from the hardware busses that the FCS uses to communicate with its peripherals. The "hierarchical" portion of the name refers to the fact that HECAD works to detect cyber attacks and faults on many levels. The levels in this case mostly pertain to the data flow to and from the sensors and the context for how the flight controller interprets and uses this information. HECAD comprises four different types of modules. Fig. 2.1 displays a generic block diagram of HECAD. At the lowest level, HECAD is required to monitor for hardware faults and attacks. The component responsible for detecting problems at this layer is named the Hardware Resource Integrity Monitor (HRIM). There is one HRIM per hardware bus used by the flight controller. This is because the component is not just responsible for detecting attacks and faults, but also capturing data being transmitted and received for the higher levels of HECAD to process. The lower bound of abstraction contained within the HRIM layer stops at logical interpretations of the signals coming in from the data bus. This means that things like current and voltage levels are not directly measured by an HRIM. It can, for example, detect if an open circuit has occurred due to a signal being inactive for a time period longer than that sensor's sampling period. The scope of attacks and faults contained within an HRIM is upper-bounded by the communication protocol. It needs to be sensor agnostic, as there might be multiple sensors that use the same communication protocol, or even on the same bus in the case of I2C. Anything related to the context of the sensor and its sampled data falls into the next level of hierarchy, the Information Integrity Monitor (I2M).

There is an I2M instance for every sensor on-board the UAS. The role of the component is to take the packaged data parsed off the hardware busses from an HRIM and detect problems, malicious or otherwise, present within. An I2M can, and in some cases require, a priori information about the sensor. This is because it will run checks like whether a returned

value falls within a valid operating range for that sensor, or detect large discontinuities within the time series of data. Like the HRIM modules, an I2M also has a role in the dataflow inside HECAD. The information returned by an HRIM is designed to be a raw packet of information without specific context. The I2M typically must first parse this raw data to assign meaning within the context of the sensor. For example, a string of binary information returned from a serial device needs to be parsed for address information or some other method of determining the data's meaning or purpose. This is not only required to perform most of the information validation required but also so that an I2M can pass this data up to the top level of HECAD.

The Functional Integrity Monitor (FIM) sits at the top of the HECAD datapath. There is only one FIM within the proposed HECAD architecture. Up until this point, each flow of information created by the individual hardware busses have had no interaction. The uncoupled nature of these data paths allow the lower level modules to make more discriminative decisions on whether a fault has occurred. It is not until the FIM that the context of the mission and the overall function of the UAS is considered. The FIM takes in all of the sensor information being presented to the FCS and can perform checks that require sensor cross-validation. These would be things like a spoofing attack that has a GPS returning monotonically increasing values in a certain direction. If the orientation and acceleration from an Inertial Measurement Unit (IMU) typically equipped on-board a UAS do not match up with the GPS data, that attack can be detected and mitigated. The FIM is presented as an area of future work in [1] and has less rigid requirements than the lower level components. It is generally conceptualized as the final stop in the data path through HECAD where all final attack or fault decisions are made.

A separate datapath exists within HECAD pertaining to the fault and attack decisions. One requirement not described above is how the components should react in the case of detecting an error of any kind. For HRIMs it is generally required that malformed data should not be propagated on to the I2M. Upon any kind of potential detected problem both the HRIMs and the I2Ms can signal to the FIM that these faults have been detected.

These error signals can be combined with the sensor data by the FIM which then ultimately makes a decision. The potential mitigation techniques were again outside the scope of the dissertation but general mitigation strategies like cutting off sensors that have been deemed faulty or infected were suggested. This decision flow should run opposite to the data coming up from the sensors. Only a higher level hierarchy should be able to control a lower level. This ensures better decision making, as the higher levels will always have more information as well as that information being more contextual.

The fourth and final component of HECAD sits outside the sensor datapath and the decision flow between the HRIMs, I2Ms, and the FIM. This is the Execution Integrity Monitor (EIM), responsible for detecting faults and attacks that do not pertain to specific sensor hardware or information. It takes a more holistic approach at monitoring the health of the hardware and ensuring the flight control software is executing normally. The implementation details again were outside the scope of the proposed HECAD architecture. Recommended options include monitoring current levels or power consumption within the FCS and comparing to known normal levels or taking trace data though the debug port of the flight control processor to ensure no atypical code execution has occurred.

## 2.3 VCU Aries Tiny FCS

The VCU Aries Tiny FCS is a fully functional flight control system developed from directed research and theses [22][23] from the UAV laboratory. It is the only flight controller architecture than HECAD is currently aimed at protecting. Aries encompasses the flight control software that features a fully functional autopilot for both fixed wing and quadcopter UAS, and the hardware design for an integrated PCB. The flight controller runs on an ARM based STM microprocessor which connects to the typical host of sensors required on a UAS. Most sensors are integrated onto the PCB but some are routed through external busses. The core functionality all sits soldered to the PCB. Several bus communication protocols are uti-

lized to communicate with the various peripherals. The GPS and GCS communication are linked over separate Universal Asynchronous Receive/Transmit (UART). There are multiple internal and external Inter-Integrated Circuit (I2C) busses connecting the barometer and airspeed in the case of the former, and a compass and current sensor on the latter. The last sensor of note is the IMU which communicates over a Serial Peripheral Interface bus. There are other devices connected to the FCS but do not currently have HECAD modules targeted at monitoring them.

## 2.4 Serial Peripheral Interface Communication Protocol

The Serial Peripheral Interface (SPI) communication protocol has existed for many decades. It is a synchronous communication standard typically aimed at embedded applications where low cost, power, and overhead are desired. It connects a master device with as many slave devices as required. At the hardware level, SPI communication requires at least four signals. A master device can send data over one of the full duplex lines typically called the Master-Out Slave-In (MOSI) signal. The master gets information back over the Master-In Slave-Out (MISO) signal. As it is a synchronous communication standard, a Serial Clock (SCLK) line must be generated by the master and shared with all its slaves. The last required signal is the slave-select (SS), also commonly referred to as the chip-select (CS). There is one CS signal per slave device. The CS is also sometimes displayed with an overhead bar to denote that by the communication standard it is an active low signal.

SPI can operate in four different modes, encoded zero through three, as displayed in Table 2.1. The mode number can be interpreted as an encoding of two different binary parameters, Clock Polarity (CPOL) and Clock Phase (CPHA), represented as the most significant bit and least significant bit respectively. CPOL determines the default polarity of the SCLK. When no communication is occurring on the bus, a CPOL of '0' (implying modes zero and one) the SCLK will be held low. A CPOL of '1' (modes two and three), the opposite will be

| CPOL | CPHA | SPI Mode | Sample Edge | Shift Edge |
|---|---|---|---|---|
| 0 | 0 | 0 | 1st Rising | 1st Falling |
| 0 | 1 | 1 | 1st Falling | 2nd Rising |
| 1 | 0 | 2 | 1st Falling | 1st Rising |
| 1 | 1 | 3 | 1st Rising | 2nd Falling |

**Table 2.1** SPI Mode Parameters

true. The CPHA parameter defines when data is shifted and sampled in relation to an edge of the SCLK. CPHA equal to '0' (modes zero and two) means the first clock edge after a transaction starts will be a data sample and the edge after that will be a shift. Whether this is a rising or falling edge is determined by CPOL. When CPHA is equal to '1' (modes one and three), the first sample occurs on the second edge of the SCLK, resulting in the opposite pairing of edge type and sample/shift operations. The SPI mode is sometimes selectable within the master and slave devices but in the case of the peripheral on-board the Aries FCS, it is statically defined as mode three.

An SPI transaction (Fig. 2.2) happens by the following sequence of steps (with mode zero). The default state with no communication requires all CS lines to be high and the SCLK line to be low. A master device can initiate a transaction by pulling a CS signal to '0' for a slave that it wants to communicate with. Slave devices usually have set standards for how much time needs to pass between events to ensure it can respond in time. The master waits a small period, typically already having set the first bit on the MOSI, then sets the first rising edge of the SCLK. What follows is dependent on the master and slave devices. Typical conventions exist, like the data first sent by the master is usually the address it wants to either read or write to, then the slave responds in the case of a read or waits for the master to write information. The read or write command can be encoded somewhere in this first burst of bits. The serial transfer of data happens on a shift/sample pattern dictated by the

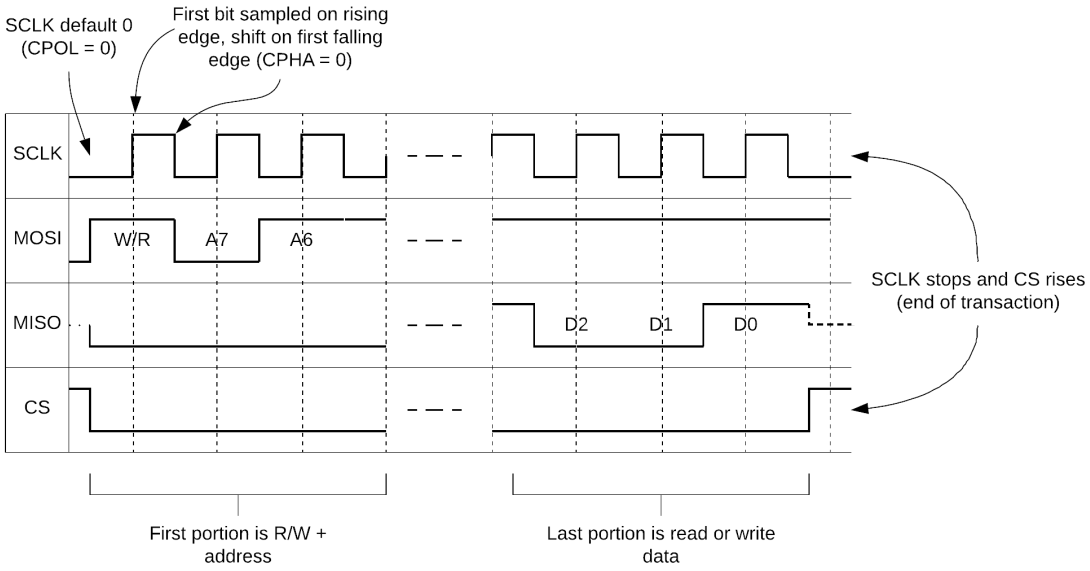


**Figure 2.2** Example Mode 0 SPI Transaction

SPI mode. In the case of mode zero, the first rising edge of the SCLK is a sample. When the clock falls back to '0', both the master and slave know to shift out the next bit in the data for it to be sampled and latched on the next rising edge. The master can end the transaction by pulling the CS signal back high after all of the data has been sent or received. There is normally a bit length boundary that the transactions must align to. Whatever atomic data unit size is present in either device dictates this. For example, a slave device with 8-bit registers and 7-bit addresses would require no less than that amount of data, plus an additional bit for R/W, transferred per transaction. SPI does support continuous, or bust-mode, operation. The master can optionally hold CS low after the minimum amount of bits has been shifted out and continue sending or receiving data. This almost always correlates to reading or writing multiple contiguous addresses, with the originally sent address serving as the base. After the transaction is complete the slave must let go of the MISO line by setting it to high impedance so other slaves could potentially respond to future master requests.

## 2.5 MPU9250

The MPU9250 is the Inertial Measurement Unit (IMU) used on the Aries Tiny FCS. An IMU offers important information to a UAS. This specific model is sampled for the gyroscope, which provides the orientation in three dimensional space, the acceleration for the same three axes, and the current temperature of the device. There is a magnetometer on this IMU, but is unused by the flight controller. This information is held in fourteen, 8-bit registers. The gyro, acceleration, and temperature sensors have integrated 16-bit ADCs, so each axis and the temperature all require a high and a low register. This information is critical for the autopilot's control system. Specifically the orientation is used in the feedback loop to ensure the system has executed whatever kind of maneuver it is attempting, even if that is holding at zero pitch. The acceleration and temperature are still important auxiliary measurements used in the FCS's control loop. For quadcopters this information is even more important, as even in manual flight, maintaining a hover would be impossible without the automated stabilization calculated from the data.

The MPU9250 features both SPI and I2C communication, though it is interfaced with SPI on-board the FCS. After Aries boots and finishes initialization and calibration, it samples all three of the sensors at 1 kHz. The flight controller does this in one continuous-mode transaction since all fourteen registers have contiguous addresses. The first bit of the SPI transaction denotes a read or a write with '0' and '1', respectively. Next the FCS sends the 7-bit address for the base data register. The CS line is held low until all fourteen registers have been read from the IMU.

# Chapter Three

# SPI HECAD Architecture

## 3.1 Overview

The SPI HECAD modules are designed to detect cyber attacks and faults for the SPI bus present in Aries and the current peripheral attached to that bus. With the current revision of the Aries FCS, the IMU sensor is sampled over the SPI protocol. Fig. 3.1 displays the block diagram for SPI HECAD. Per the requirements of the overarching design laid out in [1] and summarized in section 2.2, the HRIM and I2M are completely independent from the rest of the HECAD system besides needing a common interface to the FIM, which is outside the scope of this thesis. It should be noted that Aries does contain a flash chip that is hooked up to the same SPI bus as another slave, with its own CS signal. It is not currently used in any critical functionality by Aries so an I2M is not required to be developed.

One major design consideration for HECAD involves the coupling between the modules and the system they protect. For SPI HECAD, the level of coupling has benefits and downsides. The HRIM at least does not have to contend with this problem as much, as it is required for every bus protocol. Once developed, it does not need to be modified or updated in reaction to FCS changes, outside of completely new protocols being introduced. The I2M is predominantly the more challenging case; in order to be an effective fault or cyber attack monitor, it must have a priori knowledge of the peripheral. There is a further design consid-

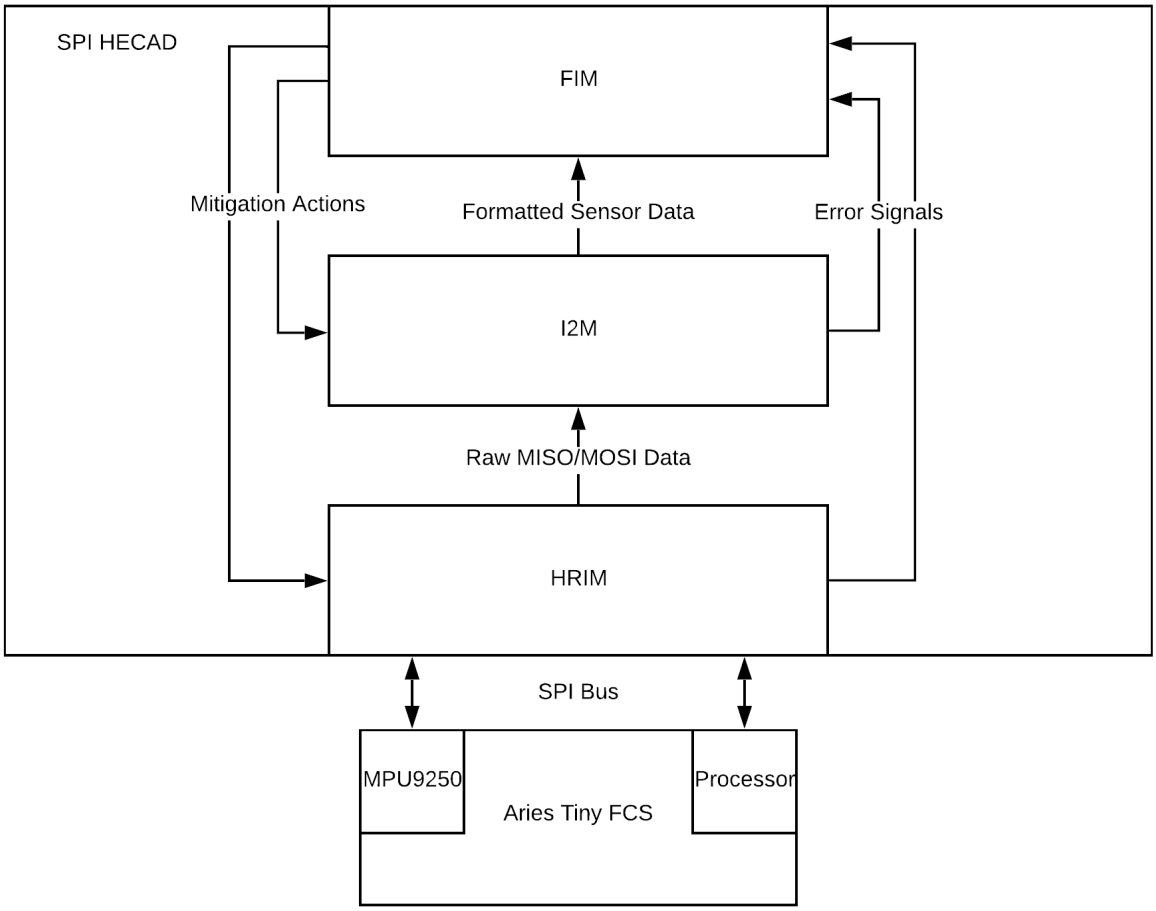


**Figure 3.1** Block Diagram of SPI HECAD

eration: the degree of coupling affects whether the I2M is only specific to the type of sensor, or both the type and model of the sensor. For example, the design choice would decide if the SPI I2M can protect a generic IMU or specifically the MPU9250. Without coupling the I2M to only the MPU9250, a high degree of pointed error checking is lost. Explicit knowledge of the sensor register map enables an I2M to validate transaction addressing and potentially stop a malicious peripheral from responding if queried outside the normal operating set of registers. Generality is certainly lost by tightly coupling the design to the sensor but without it there is very little that can be done to verify normal operation at the information level. The downside of this approach is that any updates to the FCS hardware and even software can require HECAD to be modified to match. If sensors are discontinued or have new revi-

sions, much of the corresponding I2M would need to be updated or even replaced entirely. The benefits of how powerful a monitoring device HECAD can be when implemented this way far outweighs this downside.

The SPI HRIM takes in the four SPI signals shared between the FCS processor and the IMU. It is responsible for monitoring the bus level communication to capture all of the information passing between the two devices. At startup this information is a few initializations done by setting registers in the IMU with configuration data. The gyroscope bias is also found to serve as a calibration. After this the FCS will only sample the fourteen data registers described in section 2.5. The I2M takes this raw data and determines which portions correspond to what sensor values. These modules do this as passively as possible to avoid interfering with the function of the flight controller. The SPI HECAD is only required to run detection at the HRIM and I2M levels. There should be a framework to easily extend these modules to implement error handling. As such the HRIM and I2M sit physically in between the FCS and the IMU. The detection run at the HRIM and I2M levels should follow the scope of each module as described in section 2.2. For SPI, the specific design choices for each are discussed in the coming sections, 3.2 and 3.3, for the HRIM and I2M respectively. It should be noted that these sections will contain a great deal of generalities and hypothetical situations that could exercise the following detection algorithms. The set of possible faults and attacks has far too many possibilities to specifically target and protect against. This generality is what makes HECAD more effective at protecting UASs against faults and cyber attacks. It focuses more on ensuring the system is operating within normal parameters, not attempting to detect outlying and divergent behavior.

The remainder of this chapter will cover the error classes monitored for inside SPI HECAD. Note that as SPI HECAD only encompasses an HRIM and I2M, it cannot make positive claims about the faults that lead to these errors. It is only concerned with detecting the errors and reporting them to the FIM, which will make the decision about whether a cyber attack or a natural fault has occurred and issue a corresponding mitigation action. There

will be some discussion about the potential source of these errors, as it is still important to try to capture all possible undefined behavior.

## 3.2 SPI Error Classes

The HRIM, the component responsible for detecting and reporting hardware based faults and attacks, sits at the lowest level of HECAD. It interprets the signals and ensures the transactions are conforming to the SPI standard. Within this umbrella of potential problems, the SPI HRIM breaks it down into the following three subcategories of errors. The HRIM does not make a judgement on whether the problem is malicious or a physical failure, it just reports the type of issue up to the FIM.

### 3.2.1 Signal Timeouts

The first class of abnormal behavior is classified by its dependence on signal activity. UAS software architecture typically involves a control loop, where PWM and other output signals are calculated based on the most recent sample of the current state of the system. This implies that the critical sensors in that controls system will be sampled at a regular interval. With this information, it can then be inferred that the SPI signals will have regular activity. Any signal that remains inactive for multiple samples can signify many different types of faults or attacks.

The first and most likely scenario is that a physical fault has occurred. If a trace on the PCB becomes damaged or any other kind of open circuit is created, that signal will no longer be able to propagate information. This is a very serious issue, as the loss of any of the four SPI signals will either obfuscate or completely prevent correct master and slave communication. Potential false positives for this error could take many forms. The first and most pressing problem to protect against is the system idling on startup. There must be a way to ensure that the system has entered its main control loop before starting

to check for signal timeouts. Otherwise, there is a potential problem with the SPI slave not being queried for an initial length of time that is much greater than the typical sample period. This can be circumvented by not keeping track of inactivity until some activity is first detected. If the system had already incurred a fault before startup, there may never be initial activity to set the HRIM into motion. If this happens, the FCS would likely fail in its initialization routines because it should (Aries does, at least) perform startup checks like probing the whoami register or something similar. The timeout checks are mainly there to ensure nothing has happened to the signal integrity during system operation.

Malicious actors could, if having gained access to either the FCS or the SPI peripheral, intentionally tamper with the operation of the SPI bus. This could take many forms depending on the scope of the access gained by the attacker. Any kind of focused attack on halting communication between the FCS and the SPI sensor would be detected if triggered by a condition occurring after system start up.

### 3.2.2 Transaction Errors

This class of errors has to do with how well formed the SPI transaction is. Any kind of erroneous behavior contained within a single transaction falls within this class. It is important to ensure that any data passed from the HRIM to the I2M has come from a valid SPI communication. The HRIM will already be tracing the progress of the transaction, so it can easily determine if anything has happened prematurely or did not happen at all. These types of checks will be detailed in chronological order as they could happen within a transaction.

The first type of transaction error monitored for an address alignment error. Alignment in this case refers to how data segments inside computer architectures need to have a smallest number of bits to be transferred per atomic operation. For example, for the MPU9250, addresses and registers are eight bits. No valid transaction between this device and a master

could contain a number of bits indivisible by eight. The address alignment error would occur when an SPI transaction exits before a full address has been written. This could either be from the master pulling the CS signal back up before a full address has been written or activity on the MISO signal while the master is still writing an address. An address alignment error is closer to a error contained within the purview of the I2M, as there is no strict requirement that the first portion of an SPI transaction be encoded as an address. It is however a fairly ubiquitous convention that most SPI register mapped devices abide by. This is not an issue for conforming to the requirement that the HRIM be sensor agnostic, as the next potential error is also an alignment problem that instead occurs during the data transmission. If a device does not use register mapping to determine information transfer, there are virtually no distinctions between the cases that would trigger an address or data alignment error. The FIM would be able to consider both of these error signals as equivalent, depending on its requirements.

The next potential transaction error, as mentioned, is also an alignment problem, but would occur during the data transfer phase of the SPI transaction. This is almost identical to an address alignment error; it instead would mean that something happened to cut off the full data read, which should also be divisible by an atomic number of bits. Either of these alignment errors could signify that a physical problem with the CS line has occurred and could be immediately reported to the FIM. Additionally, an attacker could want to read extra information off of an SPI device embedded after the base address but does not align on the boundary. The attacker could also be trying to intentionally hang up the peripheral by asserting more clock edges from the master before ending the transaction.

The final error detected is not caused by explicitly breaking the SPI protocol, but rather a practical limitation of either the SPI devices or the monitor. There must be a maximum number of transfers per continuous-mode transactions. The limitation can come from the master's maximum buffer size to take in this information or, even less restrictively, the entire register space of a slave device. Regardless of these two, more practically the limitation will

come from the HRIM, which can be set, with a priori information, to the largest possible continuous-mode transaction that the master device could ever perform. Aries will never sample a larger range of registers than the fourteen sensor values which can then be enforced by the HRIM. An upper limit must be set at some point, either because the master is overflowing, the HRIM's buffers are overflowing, or the slave is returning invalid data. This error is less likely to be caused by a physical signal problem, as the only way for a continuous-mode transaction to continue is for the SCLK to continue to oscillate. It is more likely that express intent has forced the master to intentionally overflow some kind of buffer. The other scenario would be an attacker attempting to read malicious information from registers padded on the outside of normal register accesses. Whatever the case may be, SPI HECAD needs to identify this behavior and report it to the FIM.

### 3.2.3 SPI Configurations

This category contains any error types that have to do with the parameters that are configurable by the master. The final hardware subclass of error have to do with the SPI modes of either the master, slave, or the HRIM. Additionally HECAD should be able to identify any combination of SPI mode configuration mismatches between any of these three components. The first configurable parameter is the SCLK frequency. The HRIM, as previously described, needs to trace the SPI transaction and capture the data in from the data signals. In order to do this, the HRIM will typically need the expected frequency of the SCLK signal that the master will provide to the peripheral. Additionally, the inclusion of this check will ensure the slave device receives a clock frequency within its specified rating.

This error could happen for a variety of reasons, though like the continuous-mode overflow error, it is more likely to happen with intentional tampering. The only way to cause this error would be for the master to set an SCLK frequency outside of the valid range for the peripheral. This maximum SCLK frequency is usually set by devices due to switching

limitations, constrained by the rating of the I/O pin buffers or the hardware implementation of the shift registers. Regardless of why the maximum frequency exists, it is almost certain that providing a SCLK that oscillates faster than specified will cause at least undefined behavior, if not outright failure of communication. This error check imposes an implementation requirement on the HRIM that it needs the ability to oversample the SPI clock. That will be discussed in the implementation section for the HRIM, section 4.5.4.

The final configuration errors have to do with the SPI mode. Mismatching the modes between the master and slave will likely prevent any communication between the two devices. These modes, described in section 2.4, are determined by two parameters. The CPOL (clock polarity) will determine the default value of the SCLK signal. Differing values between the master and slave could cause a host of different adverse behavior. The behavior will be determined by a variety of factors, including but not limited to the shift register implementation and the value of the slave's CPHA parameter. The hardware design of the slave device can vary too widely to attempt to capture these types of behavior. Differing CPHA values will be considered independently. For a slave device that is running with a CPHA of '0' but different CPOLs, the following behavior could occur. After CS is pulled down, the first sample edge will be the opposite of what the slave expects to be a sample. The implementation of the slave will determine what it does in response to this unexpected clock edge, though it is very likely to be fatal. If the slave is running with modes with a CPHA of '1', the first clock edge will match up with the type of operation, but will be offset by a clock period. This could cause alignment problems or cause the first or last bits of information to be lost.

Detection of a CPOL mismatch will depend on which device has been flipped with respect to the HRIM. The first case in which the master differs with the expected CPOL is trivial to detect, because the condition to detect this type is independent of the resulting undefined behavior. If at the exact moment after the CS signal is pulled low the SCLK signal does not match the expected default value, the error flag can be set immediately. This is because

both the HRIM and slave expect the SCLK signal to have the opposite value. The condition cannot be checked while the CS is high because it is possible the master is addressing other peripherals on the bus. The HRIM would already be monitoring these other peripherals but it is unlikely that a master is communicating with slaves over different SPI modes. Aries' SPI master only communicates with one mode, at least.

The situation is more complicated when the slave differs from the master and the HRIM. The resulting undefined behavior can take many forms. It has to be assumed that other error conditions will catch this case. Of the likely scenarios, the majority should be caught by other checks within the HRIM. If the slave device hangs because of the fault, the timeout error conditions will trip. If metastable sampling and shifting starts to occur in the slave but it continues to operate, it is extremely likely that either the address or returned data would not be interpreted correctly by either device. The I2M can be relied upon to catch this situation.

A CPHA mismatch would cause its own set of adverse behaviors depending again on the value of CPOL and the hardware architecture of the slave device. In contrast the symptoms would not depend on which device is based on a different mode than the HRIM. Generally, the clock phase being different will cause samples and shifts to differ for each device. It follows the same pattern as a CPOL mismatch. If the CPOLs are different as well, then the same clock edge is used to sample and the other to shift. This can still cause the same set of problems. For matching CPOLs, the edges will be different and can cause undefined behavior. The symptoms need to be captured in a different way, however. The HRIM can check to see if the MOSI or MISO signals have changed within too close a time period to an SCLK edge that should have represented a sample. This is the primary reason that it does not matter which device is in disagreement with its counterpart and the HRIM. Either the MOSI or the MISO will be out of phase depending on which device mismatches. The condition monitored might not only cover a mode configuration problem; there could be other faults that cause this behavior. It should still be reported as it creates metastability

problems for the master.

The last case is a situation in which the SPI devices' modes match but the HRIM has been set to expect a different mode. This is not a likely scenario. Assuming the HRIM has been maintained to keep up to date with the flight controller, the only way this could happen would be that an attacker set the modes of both devices to something different. The SPI communication would still happen normally and correctly but the HRIM would be unable to correctly interpret the transaction. Perhaps this could be done in attempts to directly attack HECAD, but that seems like a very fringe case and is not considered.

## 3.3 Information Level Error Classes

Moving up to the I2M level, there are several information checks that can be done to ensure the sensor is operating normally. The two peripherals attached to the bus that SPI HECAD monitors are the IMU, specifically the MPU9250 and a flash chip. The flash chip does not currently serve a specific purpose for Aries in its current revision and for that reason an I2M was not developed for it. The MPU9250 does have an associated HECAD module. As outlined in section 2.5, the IMU has three separate sensors sampled by Aries, the accelerometer, gyroscope, and digital thermometer. These three readings will have the same set of individual checks performed on the data, though it should be noted that for the remainder of this section they will be referred to as a collective. If a sensor has outlying conditions it will be specifically mentioned.

### 3.3.1 Invalid Addressing

This check is listed first in the section because of its proximity to a hardware error. Invalid addressing is classified as any MPU9250 address requested by Aries that has one of several potential issues. The first check performed on this information is read/write protection. The MPU9250 expects the first bit of the address segment to denote whether the command is a

read or a write. There are registers in the peripheral that have access protection and as such, cannot be written to. The fourteen data registers, for example, should never be written to by the master. The next is ensuring the address is something Aries has the normal capability to interact with under normal operation. Via the Aries MPU9250 c code, there is a known set of registers that will ever be read or written. If the address falls outside of this set, SPI HECAD can definitively assert a problem with the FCS. This error could have many potential sources. A bit in the address could have been flipped accidentally from a metastability issue or it could indicate a coordinated attempt between an infected FCS software and compromised peripheral to communicate. For example, the MPU9250 can act as a I2C master to other peripherals. While operating in SPI mode, much of this functionality is disabled as they share the same physical I/O ports, so no interaction between Aries and these I2C registers should occur. Malicious information could be stored inside these unused registers and exercised by accessing them during Aries runtime. For this reason, SPI HECAD will ensure that all transactions will have a validated address before allowing it to continue.

### 3.3.2 Stuck-At Errors

The first and simplest form of error checked for in the I2M is the stuck-at error. This error classification refers to any kind of physical failure or attack that would cause the information returned to permanently stay at one value for the rest of operation. There is a traditional interpretation of this error class in communication and other bit oriented systems where the "stuck-at" refers to a single bit position. The I2M in this case is more concerned about the entire data value as a whole instead of individual bits. There are many reasons for this, the first being that the I2M does not care about that level of granularity for error detection. Implementing stuck-at error detection on the bit level would over-fit the detection model. The context of the information would require each bit to have its own threshold for what could be potentially a stuck-at error. The data in this case is a sensor reading. Although

UASs can have incredibly fast dynamics, the more significant bits would not be subject to constant change. The temperature sensor, for example, would normally vary by a couple of degrees or fewer per sample. This means that the upper bits would almost never change throughout the duration of operation. An I2M still needs to monitor these values however. Electrical and vibrational noise will always cause a sensor to jitter around the absolute value. Kalman filtering is performed on Aries to remove this noise, as it cares more about a useful value to input to the controller software. SPI HECAD can use this noise as an accepted assurance that a correctly operating sensor will have variance in its readings across multiple samples. The lack of noise is indicative of a problem. There will be separate thresholds for what could potentially be a problem depending on the type of sensor. Considering the temperature sensor again; it will change significantly less often than the readings of a reasonably precise accelerometer, especially during flight. Each sensor will have its own threshold of how long a single value could be held before it signifies a potential problem.

The granularity at this level is more suited towards what can reliably detect information problems. Stuck-at errors for the entire reading can be caused by a number of faults. Physical faults are the most likely, something happening to the sensor hardware could cause it to be unreachable by the sensor's on-board controller. The last value present in the data registers for that sensor would then be continuously sampled by the FCS, making it appear to have never changed. This can have drastically fatal effects on the system, as there would no longer be any valid feedback to the controls system. A cyber attack, especially one within the supply-chain class, could embed malicious functionality within the sensor. The attacker, with unfettered access to the inner workings of the sensor, could permanently fix values in the registers. This would likely be detected early in the hardware design process of the flight controller, so more sophisticated tampering would be required, like some kind of time delayed or condition triggered attack. Regardless of the root cause, detecting if any of the three sensors sampled on the IMU is an important way to ensure some kind of base functionality.

### 3.3.3 Range Checking

Range checking is one of the most tightly coupled design elements present in all of HECAD. This information check requires specific knowledge of both the ratings of the sensors and how they're configured. Many sensors, such as the accelerometer in the MPU9250, have selectable ranges of measurements. For example, each quantum of the 16-bit ADC can represent the subdivision of -2 to 2, -4 to 4 G's, or whatever other range the sensor is capable of. The other piece of information required to perform range checking is the absolute maximum rated value that the sensor can reliably return. This can take on many forms as the maximum and minimum of a sensor can be caused by different factors. For the sensors in the MPU9250, all values within any of the selectable ranges can be represented by the ADC. That is, there is no raw binary value that would correspond to a reading outside the range of the sensor. For example, the maximum value for the accelerometer is the maximum signed 16-bit number returned from the ADC.

This makes range checking for this specific application not so much about ensuring the sensors are operating within their own ratings, instead about ensuring the value is not being cut off by the set range. Aries configures both the accelerometer and the gyroscope to ranges less than the maximum, $\pm$4g out of $\pm$16g and $\pm$1000 degrees per second out of $\pm$2000, respectively. This means that if the sensor measures a value outside of this range it will report to a minimum or maximum encoded value. The condition for triggering the range error signals will then be when a sensor is above or below threshold values close to the signed minimum and maximum 16-bit numbers. This will protect against several factors. If a supply-chain or any other kind of cyber attack has caused the sensor to report values around its extrema, the I2M should detect this atypical behavior. If the system is actually experiencing the forces reported around these extrema, then likely the aircraft is experiencing some serious mechanical problems and this situation should be reported anyway. It likely would not matter too much to the control system if the aircraft is making almost three full

aileron rolls a second (roughly 1000 degrees per second) versus four or five. The information being clipped is an issue and should be addressed, but it is likely there are larger and more pressing issues than the controller not receiving fully accurate measurements.

The temperature sensor is different from the other two in that it does not have a selectable range. It is worth noting again that this temperature reading is for the MPU9250's internal core, not ambient or inside the aircraft. Combined, this makes the temperature sensor a special case that makes its range check take on a different meaning. While this is open to interpretation, the strongest case can be made for setting the valid "range" to the temperature ratings for the other sensors. Both the accelerometer and the gyroscope deteriorate in accuracy as temperatures move towards the extremes. It is for this reason that the temperature sensor range check will trigger when the values sampled approach these corners.

### 3.3.4 Reasonability Checks

Range checks, as discussed in the previous section, are more likely to catch completely fatal errors and other attacks. As mentioned, by the time the gyroscope is reporting at the maximum of its range, the aircraft is performing almost three complete rotations about any of its axes. The ground control would very likely want to know about this behavior the aircraft reaches this point. That is the purpose of the reasonability checks, it is closer to the function of a warning message in a compiler rather than a complete error. The thresholds should be set in a way that if the aircraft ever starts to experience behavior outside of what the flight controller should be able to perform, the reasonability check signals will fire. The FIM will have to take into account the varying severities of the incoming error signals from the SPI I2M, along with the other I2Ms. Checks like these can give a forecast that erroneous behavior is about to occur, at least in the case of the acceleration, orientation, and temperature. For sensors like GPS, it can be assumed that if the aircraft started in the northern hemisphere, the GPS should never display the aircraft in the southern hemisphere.

Such overt reasonable values do not exist for physical phenomena in such a clear-cut way. For this reason, the threshold values should be set somewhere between the range checking limit and what could be considered just outside normal operation. The inner bound might need to take experimental data into account if the expected normal operation is not overtly available from the flight control code.

### 3.3.5 Discontinuity Checks

The final type of monitoring performed in the I2M is a discontinuity check. There are many ways to interpret what a discontinuous sensor reading entails. For the IMU I2M, a discontinuity will be considered as a current sensor sample that has a delta value outside of a threshold. Any jump in a sensor value that lies significantly outside where the previous samples have pooled is cause for alarm. Like the reasonability checks, discontinuity is not necessarily indicative of a failure but still warrants reporting to the FIM. The magnitude of the jump will also affect the severity of the problem and can correspond to separate error or warning signals. For example, a change in 20 degrees Celsius of the IMU within a single sample period is very alarming but not physically impossible. It does not have to always indicate a failure but it very well could. Almost a full swing across the temperature range, say from 30 degrees up to 95 degrees (within a 1kHz period) would almost always be caused by a serious fault.

## 3.4 Error Signals

At this point the FIM has the ability to combine multiple HRIM and I2M error and warning signals to more definitively diagnose problems. The various levels of severity implicit within the different checks performed should allow the FIM to discriminate between potential problems and fatal errors. Some signals on their own will always signify catastrophic system failure if allowed to continue to propagate. These, like the range checks in the I2M or any of

the hardware checks, should have a corresponding level of mitigation action taken to protect Aries and the aircraft from further damage. Others, like the reasonability and discontinuity checks can suggest problems but not always indicate future or ongoing failures. The FIM would have to take multiple warnings into account before issuing a mitigation strategy.

For example, if a discontinuity is identified then a stuck-at error occurs, the FIM can reasonably assume that the sensor has experienced a fatal error. The discontinuity itself would not trigger a mitigation action, such as hot-swapping to a backup IMU, but still provides valuable context to the FIM. A related situation where all three IMU sensors indicate a discontinuity within the same sample could be responded to much faster than waiting for the signal timeout. The signal timeout would then inform the FIM the reason for the error and could adjust mitigation strategies. If the backup IMU was running on the same SPI bus, the shared MOSI/MISO lines that have timed-out will not allow the system to rely on the backup.

While the FIM lies outside the scope of this thesis, the development of SPI HECAD should keep this context in mind. The architectural decisions need to be matched across HECAD modules for other busses and other sensors so that the FIM can be presented with a consistent interface so that it can make these high level decisions.

# Chapter Four

# Implementation

## 4.1 Overview

This chapter will cover design decisions' justification of SPI HECAD and then how it is implemented. There are several factors that affect how best to implement SPI HECAD to ensure that it holds both the requirements laid out by the overall architecture presented in Dr. Leccadito's dissertation and the specifications from Chapter 3. The entire HECAD system is implemented in a Xilinx System on a Chip (SoC). This hardware platform combines the flexibility and power of an FPGA with the convenience and accessibility of an embedded processor. A discussion on the hardware/software partitioning of SPI HECAD can be found in Section 4.2.1.

The entire HECAD design is tightly integrated with the Aries Tiny FCS, requiring both close access to all of the internal busses on the flight controller PCB and information about the peripherals and the FCS code responsible for interfacing with said peripherals. The information SPI HECAD specifically requires about Aries can be found in Section 4.4. A revision of the Aries FCS will combine it with the HECAD platform into one hardware design. This design, while outside the scope of this thesis, is worth mentioning as it heavily affects the design choices for SPI HECAD and is the operating environment for the implementation details presented in this chapter. The design will allow all of the HRIM modules to have

direct access to the hardware signals passing to and from Aries and all of its peripherals. The signals have to pass through HECAD in either direction. The details of this datapath and the monitoring mechanisms are presented in sections 4.5 and 4.6.

## 4.2 Hardware Platform

### 4.2.1 Hardware/Software Partitioning

The production of Xilinx SoCs have greatly simplified the hardware/software partitioning problem. Partitioning involves deciding which portions of functionality of the design should be implemented in hardware, typically an FPGA, and what should be programmed into a sequential processor. An SoC mitigates the rigidity of these decisions by combining both platforms into a single silicon chip. Segments of the design still need to be placed in either domain but the SoC allows the combination of the two much easier, especially with Xilinx's design tools.

The two main deliverable components of this thesis, the HRIM and I2M for SPI and the IMU, respectively, are both implemented in VHDL, meaning that they will reside in the FPGA portion of the design. Many reasons for this are shared from the main justification of HRIMs and I2Ms being implemented in hardware. There are many completely independent parallel data paths flowing through the entire HECAD system. Given that this is an embedded design, size, weight, and power (SWaP) constraints are very important to consider. There are no commercially available microprocessors that would meet the SWaP constraints for HECAD because of its highly parallel workload. The HRIMs and I2Ms for every bus and sensor data path need to be available for processing as soon as bus activity occurs. Multicore CPUs might be able to keep up with this requirement but will not meet the SWaP constraints. Furthermore, HECAD requires extremely low latency response times to the bus activity down to the bit level. For immediate error mitigation or prevention techniques to

function, HECAD needs to be able to react as soon as atypical behavior occurs. For this reason, at the very least the SPI HRIM needs to be implemented in hardware. The SPI I2M could potentially be moved into the software domain if it were the only I2M present in the system. Latency is still important, but its error and attack detection run at the sample level. The SPI I2M is not the only module at this level present in the whole HECAD system, however. The parallel nature of this workload also requires the I2M to be implemented in hardware.

### 4.2.2 Zynq Ultrascale+

The specific Xilinx SoC architecture selected for HECAD is the Zynq Ultrascale+. There are multiple tiers of SoC architectures within the Zynq family. The Ultrascale+ MPSoC line of products are at the top of these tiers. Development began with the originally implemented modules from Dr. Leccadito's dissertation. It was found that original hardware platforms were not sufficiently large enough to contain the entire design once implemented. The specific issue was the amount of programmable logic (PL) in the original hardware platform. The PL in a Zynq refers to the FPGA fabric. A new board, specifically the Avnet UltraZed EV and its carrier card, was selected because of its ample amount of fabric and its improved processing system (PS), which in Zynq terminology refers to the embedded microcontroller responsible for running any sequential workloads.

A simplified block diagram of the Zynq architecture as it relates to SPI HECAD can be found in Fig. 4.1. The PS and PL communicate over a high-speed, low latency bus named AXI. This interconnection is what makes the Zynq architecture so well suited for HECAD. The HRIMs and I2Ms in the PL have a reliably fast way to communicate with the PS, which in this case will contain the FIM. The EIM, though not currently implemented, can reside in either domain depending on how it is implemented. The PS has the ability to communicate with many separate modules in the PL with the use of Xilinx's AXI IP Workflow. In order to

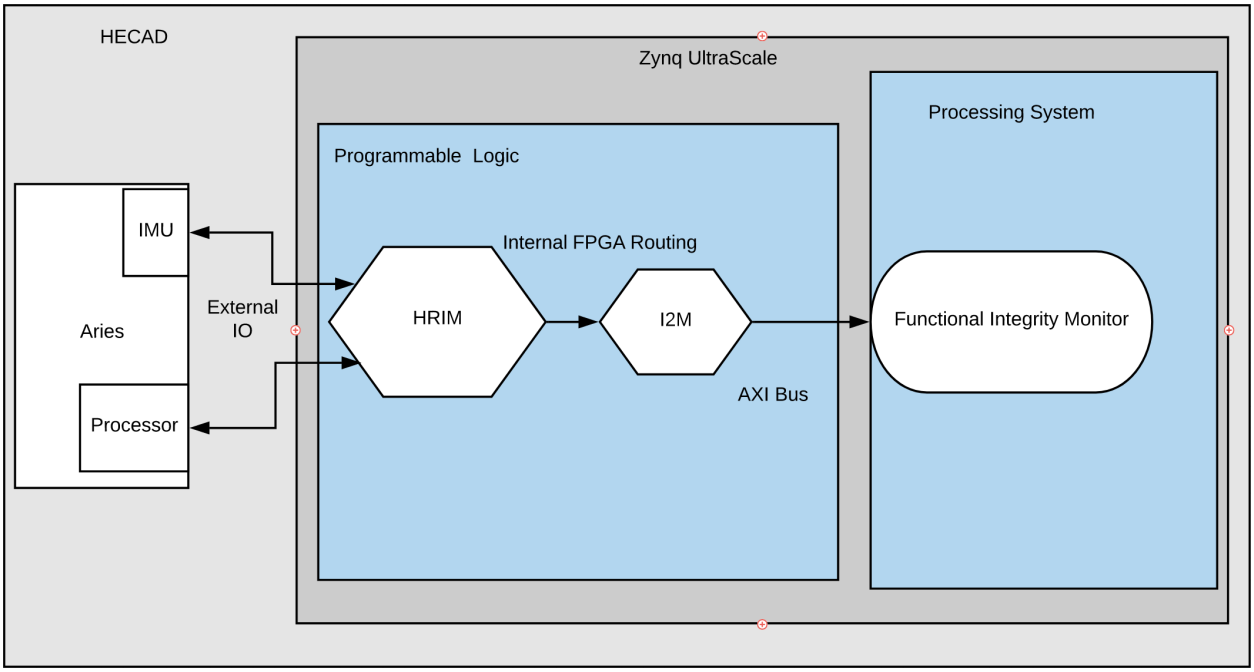


**Figure 4.1** SPI HECAD Within Zynq Architecture

communicate with the AXI bus, any modules, the HRIMs and I2Ms in this case, need to be packaged as an AXI peripheral. This process is very straightforward and easily implemented. AXI peripherals can be set in master or slave mode; for SPI HECAD's purposes, the modules will be configured as slaves. This gives them access to a configurable amount of registers which are used as the primary method of communication across the AXI bus. AXI slaves are typically memory mapped and will be in this case as well. Interrupts are supported in both directions, so the PL peripherals will have the ability to notify the FIM whenever issues are detected or sensor data is ready for processing.

The first revision of Aries that contains HECAD will utilize the UltraZed's carrier card. The UltraZed is a System on Module (SoM), but is sold with a carrier card that greatly increases the systems I/O capabilities and is ready for use out of the box. Using only the SoM would require more complicated PCB design, so the decision was made to attach a modified Aries PCB that breaks out all of its internal busses to a FMC HPC connector to the UltraZed carrier card which features the same type of connector. The FMC connector

on the carrier card can be fed directly into the PL, which makes it ideal to easily route the signals to the corresponding HECAD modules.

## 4.3 Datapath

One primary function of SPI HECAD, as discussed in the previous chapter, is to parse information passing to and from the FCS and the IMU. With the modified Aries, the required signals will be input to the PL through the FMC connector. For SPI HECAD, the four SPI signals are all required to pass in and out of the design. The following naming convention will be adopted for designating the direction of these signals: the prefix "M_" will be appended to signals that flow to or from the SPI master. For example, the M_MOSI signal is the normal MOSI line driven by Aries that is intercepted by SPI HECAD. The M_MISO signal is the MISO signal passed through SPI HECAD on the side of the master. The opposite is true for signals with "S_" appended to them. They are all attached from SPI HECAD to the IMU; so S_MOSI is the signal coming from SPI HECAD and going to the IMU. Similarly, the S_MISO signal is being output from the IMU and intercepted by SPI HECAD. Fig. 4.2 displays this naming convention, as well as the highest level block diagram of SPI HECAD. The four SPI signals can be seen on either side of SPI HECAD communicating with the respective device.

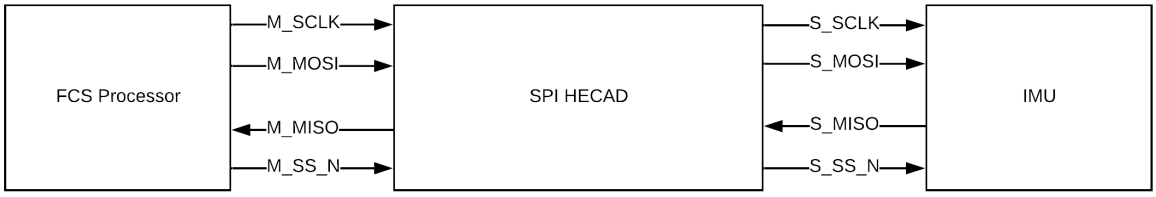


**Figure 4.2** SPI Signal Naming Convention

Zooming into the SPI HECAD block, a more detailed overview can be seen in Fig. 4.3.

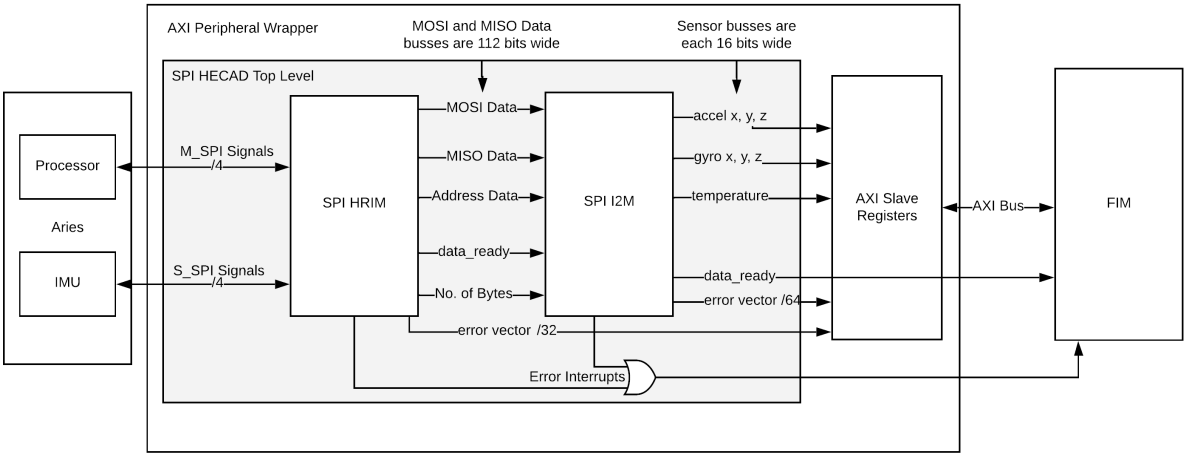


**Figure 4.3** Detailed Block Diagram of SPI HECAD

Note that from this point forward and further use of the terms HRIM and I2M will be in reference to only the SPI variants of these modules unless otherwise stated. At this level, the interconnection between the HRIM and I2M can be seen as contained within the SPI HECAD top level module, which is packaged as an AXI peripheral. The HRIM takes in the SPI signals from the flight controller and the IMU and looks for activity on the bus. The entire transaction is captured and stored into registers for the address, MOSI, and MISO data that was exchanged. The SPI signals are passed through to the IMU unperturbed under normal operation, as SPI HECAD is a passive monitor. The signals can be rerouted if an error occurs and the command to do so is received by the HRIM. The packaged SPI transaction information is sent to the I2M over the corresponding signals. The HRIM informs the I2M that data is ready with a pulsed signal. The I2M captures this information into its own registers so that it can parse which MPU9250 registers were read from or written to. The I2M decodes this information by checking the provided address and sends it to AXI registers contained within the AXI wrapper. At this point, if the data has been validated by the I2M, the module will send an interrupt to the FIM informing it that data from the most recent SPI transaction is available for pickup across the AXI bus.

The sensor readings datapath terminates at the FIM where it can make judgements with

all of the sensor information available to it. As the FIM is outside the scope of this thesis and is not currently implemented, an interface is not present in either the HRIM or I2M to receive error mitigation actions. Upon further development of the FIM, the SPI HRIM and I2M will need to be extended for this functionality. The implementation of the HRIM and I2M is done in a way to make this extension as trivial as possible. Error reporting framework is implemented and too is displayed in Fig. 4.3. Both the HRIM and I2M have a dedicated port for reporting errors up to the FIM. Each port is an encoded vector, with each bit corresponding to a type of detected fault or attack as described in Chapter 3. More detailed information on the error reporting can be found in each of the HRIM's and I2M's respective implementation sections, 4.5 and 4.6. Both modules output their error vectors and an interrupt to the top level. These error interrupts are logically ORed in the top level and sent directly to the FIM. The error vectors are stored in AXI registers that can be read by the FIM upon receiving an error interrupt.

## 4.4 Design Coupling

There has already been discussion on why the SPI HECAD modules need information about Aries in the previous chapter. This section will discuss what parameters SPI HECAD specifically needs embedded into the design. For ease of use and extensibility, a VHDL package was created containing all of the necessary parameters and values required by SPI HECAD. More detailed information about how these are used in SPI HECAD can be found in the HRIM and I2M implementation sections.

Starting with the SPI parameters within the HRIM, the first is the SPI mode. It should be noted that only modes 0 and 1 are supported by these generics, i.e. only the SCLK phase will matter. Modes 2 and 3 were not implemented for the sake of design complexity and because both the flight controller and the MPU9250 are locked to mode 0. These are required because the HRIM needs to understand how to interpret the SPI signals input to

the system and encode the information being passed on the bus.

The HRIM also requires several frequencies input to the VHDL package. Used in several calculations, the SCLK frequency is required so that the HRIM can correctly parse the SPI transaction. It is also used in ensuring the measured SCLK signal is running at the correct frequency. The HRIM needs the system clock frequency as well so that it can determine the ratio between the two. SPI HECAD also requires the sampling frequency of the IMU, or how fast the flight controller requests the sensor data from the IMU. Without it, the HRIM will not be able to determine how long signal inactivity should register as a timeout.

The last pieces of information required by the HRIM are the address length, register data length and the size of the maximum possible continuous-mode transaction. The address length will only be used in SPI devices that are memory mapped. Register data length is self evident, though for non-memory mapped devices this can be thought of as the smallest possible transaction length. Non-memory mapped SPI devices are rather rare and were not considered in the development of SPI HECAD. The HRIM does require the maximum possible continuous mode length so that internal vectors can be set to capture the information from the MOSI and MISO signals. Practically, there will always be a limit constrained by the master or slave device. This value can be set based on how the master is programmed instead of the maximum number of registers in the SPI device. This will save FPGA resources and allow the continuous-mode overflow error detection to be more powerful.

The remainder of the contents in the SPI HECAD VHDL package are used as error condition thresholds. These values are required by SPI HECAD to accurately detect problems within the system, but they are not parameters objectively obtained from Aries, its programming, or the SPI peripheral. For example, the number of samples that a signal must be inactive before its timeout error fires can be set here. These are pulled out into the package for ease of use, should future development of HECAD or the specific SPI modules require tuning of these thresholds and parameters. The remaining types of thresholds are not listed here, though any value required by the error detection methodology described in Chapter 3

will be present in the VHDL package.

## 4.5 HRIM

### 4.5.1 Component Description

The SPI HRIM is composed of several finite state machines (FSMs). As displayed in Fig. 4.3, the HRIM takes in the four SPI signals from the flight controller and IMU. M_MOSI, M_SCLK, and M_CS are all signals input or output by the flight controller processor. This naming convention was chosen to more easily differentiate which device SPI HECAD communicates with over these signals. The IMU signals are all present, instead with the S_ prefix. The main outputs of the HRIM are the transaction data pulled from the bus, along with a data_ready signal to notify the I2M when a transaction has completed. The output error vector and interrupt signals are sent to the top level for assignment in the AXI wrapper.

One design decision warrants more specific discussion. Dr. Leccadito originally proposed that HRIM modules be developed from COTS intellectual property (IP) cores. The thought behind this is that HRIMs primarily serve as an interpreter for the bus protocols they monitor. This idea was explored when implementing the SPI HRIM. SPI unfortunately does not lend itself to this method as simply as UART or I2C. After searching for a product that could function at least partially as an interpreter, this idea was abandoned in favor of writing an HRIM from scratch. Primarily the issue with SPI COTS IP cores is that the data sampling functionality is typically intertwined with the output. Using an SPI master alone does not satisfy the requirements as it will generate its own SCLK and CS signals. An SPI slave IP core almost always has its own register space and its own way of address decoding. As SPI is not a very complicated communication protocol, the HRIM was built up from scratch instead.

### 4.5.2 Clocks Domains and Asynchronous Inputs

There are two main clock domains within the SPI HRIM. The system treats the SCLK line as a clock signal. It creates some constraints however, as this creates the first clock domain, while the second runs off the system clock. Note that the system clock refers to whichever fabric clock is provided to the SPI HECAD AXI peripheral.The SPI clock domain is primarily responsible for shifting in data to the HRIM buffers. Treating SCLK as a clock signal allows the use of the rising_edge VHDL function. This makes shifting data significantly easier than it would be otherwise. The rest of the HRIM falls within the system clock domain. Managing two clock domains with such a high degree of coupled functionality requires special treatment of some classes of signals signals. The first class is the system resets.

```
258     ------------------------------------------------async reset sync-----------------------------------
259
260     --First component takes reset and syncs to system clock
261     xpm_cdc_async_rst_inst : xpm_cdc_async_rst
262     generic map (
263       DEST_SYNC_FF => 4,    -- DECIMAL; range: 2-10
264       INIT_SYNC_FF => 0,    -- DECIMAL; 0=disable simulation init values, 1=enable simulation init values
265       RST_ACTIVE_HIGH => 0  -- DECIMAL; 0=active low reset, 1=active high reset
266     )
267     port map (
268       dest_arst => sync_reset_n, -- 1-bit output: src_arst asynchronous reset signal synchronized to destination
269                                  -- clock domain. This output is registered. NOTE: Signal asserts asynchronously
270                                  -- but deasserts synchronously to dest_clk. Width of the reset signal is at least
271                                  -- (DEST_SYNC_FF*dest_clk) period.
272
273       dest_clk => clk,   -- 1-bit input: Destination clock.
274       src_arst => reset_n    -- 1-bit input: Source asynchronous reset signal.
275     );
276
277     --second component takes the synced reset from above and syncs it again to the SPI clock, creating an SPI Reset
278     xpm_cdc_sync_rst_inst : xpm_cdc_sync_rst
279     generic map (
280       DEST_SYNC_FF => 4,    -- DECIMAL; range: 2-10
281       INIT => 1,            -- DECIMAL; 0=initialize synchronization registers to 0, 1=initialize
282                             -- synchronization registers to 1
283       INIT_SYNC_FF => 0,    -- DECIMAL; 0=disable simulation init values, 1=enable simulation init values
284       SIM_ASSERT_CHK => 0  -- DECIMAL; 0=disable simulation messages, 1=enable simulation messages
285     )
286     port map (
287       dest_rst => spi_reset_n, -- 1-bit output: src_rst synchronized to the destination clock domain. This output
288                                -- is registered.
289
290       dest_clk => sclk_sig, -- 1-bit input: Destination clock.
291       src_rst => sync_reset_n    -- 1-bit input: Source reset signal.
292     );
```

**Figure 4.4** Handling Asynchronous Resets Through Xilinx VHDL Templates

Fig. 4.4 displays two Xilinx VHDL templates required for reset synchronization. The first takes the reset signal provided by the system and synchronizes it to the system clock. In its current implementation, SPI HECAD receives an asynchronous reset from a button on the FPGA for ease of development. This component is not required if the reset provided by

the system is already synchronous with the system clock. The second component is similar to the first, but instead takes an already synchronous reset (the output of the first component) and is used to create a second reset signal, synchronous to another clock domain. This component is responsible for providing the reset to the SPI clock domain.

The second class of special case signals are the SPI signals. These are input from outside the FPGA, so the system clock will not be synchronous to these signals. All of them are used in the data and control path state machine. Many problems arise from using asynchronous inputs to an FSM. The first and most problematic is nondeterministic behavior during state transitions. Without synchronization, these signals will cause the state registers for FSM to go to a state not encoded by the state variables. The state machine reaches a terminal state in this situation and can not recover without external reset. A transition can sometimes not be taken or be taken prematurely. This adverse behavior requires the synchronization of the SPI signals to the system clock. Sampling them inside a clocked process will ensure that changes will only occur on clock edges, making them safe to use for state machines.

```
302
303     mode <= cpol XOR cpha;  --'1' for modes that write on rising edge
304     WITH mode SELECT
305     sclk_sig <= m_sclk WHEN '1',
306             NOT m_sclk WHEN OTHERS;
307
```

**Figure 4.5** Allowing the HRIM to be Generic to SPI Modes 1 and 2

One thing of note is how the HRIM handles SPI mode configurations. As mentioned above in Section 4.4, the HRIM is only configurable to modes 0 and 1. The way the HRIM handles this is by interpreting the clock signal in accordance to the mode. Fig. 4.5 displays how this is performed in the VHDL. The result of the XOR operation between the two mode parameters allows the differentiation of the clock polarity. For modes 1 and 2 (CPOL XOR CPHA = 1), the SPI modules will write on the falling edge of the SCLK signal. For modes 0 and 3, the opposite will be true. The HRIM can be generic between modes 0 and 1 because

all that is required to translate the two is inverting the SCLK signal if mode 0 is selected. The falling edge can then always be used to capture data. This will be used in the remainder of the HRIM, so it is important to note that all functionality will be based off of this inverted clock for mode 0.

### 4.5.3 Data and Control Paths

This subsection will cover how the main data and control path (DCP) FSM functions inside the SPI HRIM. As with most communication protocol interpreters and devices, the SPI HRIM requires an FSM, as there are separate phases of the transaction that depend on past inputs. This FSM is primarily responsible for following along with the SPI transaction and parsing the information from the data lines, then packaging the data for shipment to the I2M. There are error detection components present within the states in the DCP FSM but discussion of these portions will be reserved for Section 4.5.4.

Fig. 4.6 displays the entire state transition diagram for the DCP FSM. Extended state machine notation is adopted for this diagram. Note that this is also a hierarchical state machine. The superstates will all have two substates, each with the root name appended with "_high" or "_low". Generally the naming convention refers to the "_high" substate waiting for the corresponding control signal to be zero, and "_low" waiting for it to be one.

The first state is wait_low, where the FSM will wait for the SS_N (active low slave/chip select) signal to come back up to high. If the state machine starts on waiting for the SS signal to drop to initiate an SPI transaction, there is a change if Aries has already started up, that the HRIM will begin interpreting the data and hang. Generally SPI HECAD will always be running before Aries boots, but this ensures if the HRIM needs to be reset it can handle monitoring the bus when it is in any state. After any first partial transaction has finished, the FSM transitions to wait_high. Here, the system is idle until the flight controller initiates a transaction by pulling the CS signal low.

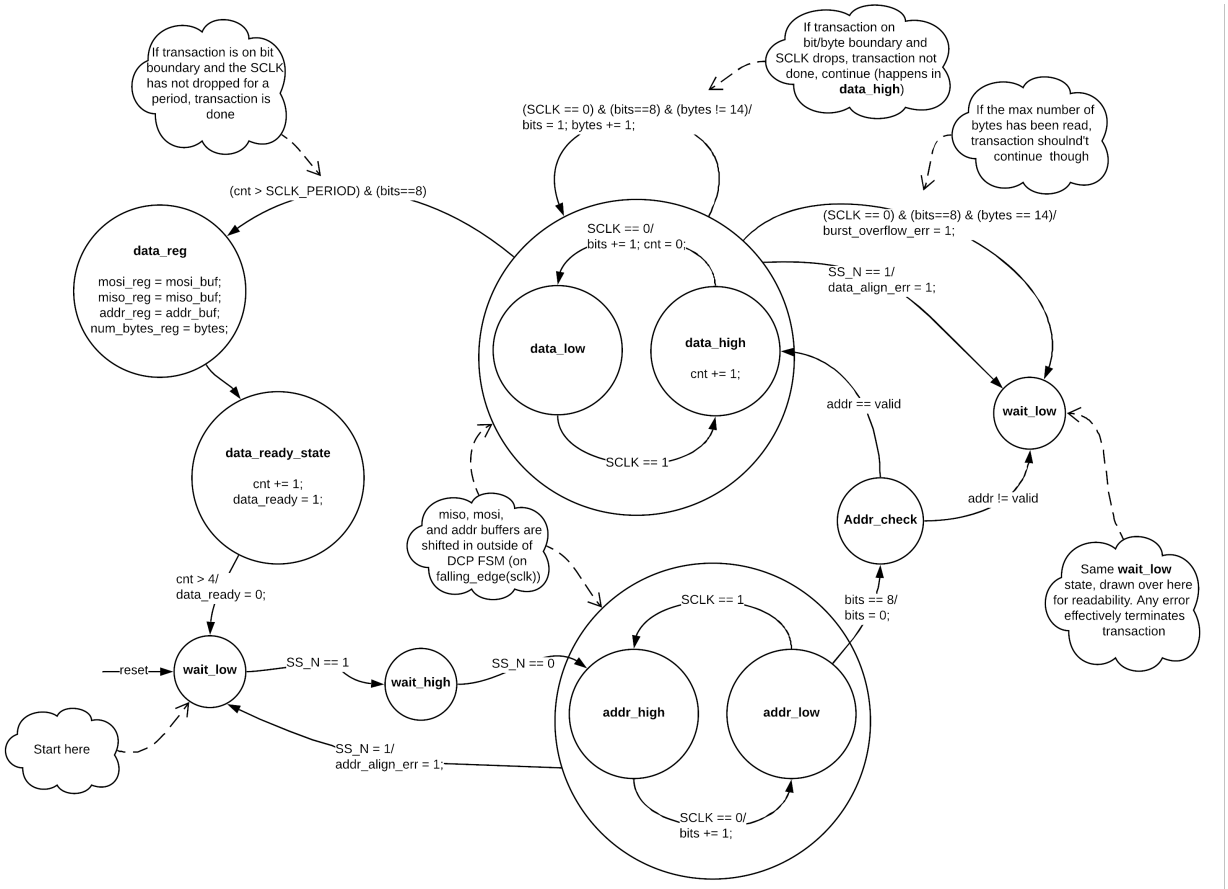


**Figure 4.6** Data and Control Path Finite State Machine State Transition Diagram

The DCP FSM now reaches the address superstate. This is a reset transition, so it will always enter into the address_high state. The first SCLK periods, based on the address length constant, of an SPI transaction in this system are the R/W bit and the 7 address bits. The FSM will transition back and forth between the "_high" and "_low" substates for every SCLK edge. The outputs of these two states are displayed because in the VHDL, they do not capture the data. There is a separate process which takes these state variables as enable signals for the shift registers. This will be described in more detail later, but note that this also applies for the data states as well. The case to transition out of the address super state occurs when a counter, incremented every time the FSM leaves the address_high state. This "bits" counter keeps track of the number of SCLK periods that have occurred.

In the address_low state, upon the detection of a rising edge of SCLK, this counter will be compared against the address length. If it is equal, that signifies that the address portion of the transaction is completed. The FSM transitions to an address error checking state when this condition is met.

The address checking mechanism was implemented into the HRIM, despite it being within the domain of the I2M. Potential error mitigation strategies can be implemented in this state. If an invalid address is detected, it's possible for the HRIM to stop the FCS from receiving the data in real time. The I2M does not receive any data until the HRIM has read in the entire transaction, so in order to perform this type of mitigation strategy, the check needs to be performed here instead. The address_check state will simply ensure the address is within the specifications laid out in Section 3.3.1. If no error conditions are found, the FSM will transition to the data superstate.

The data superstate is very similar in structure to its address counterpart. There are two substates, data_high and data_low. These similarly transition back and forth on edges of the SCLK signal. The states are again used as enables to shift in the transaction data, not displayed in the state transition diagram. The data_high state has significantly more complex transition logic, however. Continuous-mode operation requires more consideration. To explain, several scenarios will be presented. The first scenario is that the SPI transaction is a single register read or write. In this case, the transaction is done after the eight data bits are shifted across the bus. Then, after the final bit is shifted, the SCLK will have just returned to the default state (low for mode 0). This alone is not significant enough to denote the end of the transaction; it is not until the CS signal comes back high. This scenario is easily implemented, as it follows the exact same logic of the address superstate. The added complexity comes from the following scenario. In this situation, the SPI transaction will run in continuous mode for an indeterminate amount of time. After a single register is read or written, the FSM must determine if the transaction is done. Similar to the address super state, another counter is tracking the number of bits shifted so far in the transaction. When

this reaches the register data length, there are two possible conditions. The first, like scenario one, is that the transaction is finished. The second, like scenario two, is that there is more information about to be shifted across the data signals. In order to capture the continuous mode operation, the DCP FSM will check for another SCLK edge. If it does detect an edge, then another counter is incremented keeping track of the number of bytes read or written in the transaction. If an amount of time passes that would exceed an SCLK period, the FSM can definitively transition out of the data superstate, as the transaction has completed.

The DCP FSM then transitions to a state used to capture the transaction data into separate registers. This is done so that the I2M is presented with consistent data across the shared port signals. The shift buffers used to capture the data in the HRIM would change across the transaction, so separate state registers are used so that the I2M does not have a time constraint to use the information before it changes. The contents of the address, MOSI, and MISO shift buffers are copied over, along with the number of bytes present in the transaction. The number of bytes is required because the registers used in this operation are set to the maximum possible length that a continuous mode transaction could transfer. The number of bytes allows the I2M to determine how much of the MOSI and MISO registers is valid data, not padded zeros. The data register state waits for a few system clock cycles and is used as an enable for these static registers.

After the final data is captured, the FSM moves to a state responsible for signaling the I2M that a new transaction has occurred. The data_ready state sets a signal for a few system clock cycles to ensure the I2M can capture it. The FSM loops back around to the idle waiting states to check for further transactions. This concludes the main DCP FSM functionality.

### 4.5.4 Error Checking and Reporting

The HRIM detects all of the potential problems laid out in Section 3.2. The three classes explained there are implemented in separate portions of the VHDL description. Each type of error has its own one-hot encoded signal that is output through the portmap of the HRIM. Within the VHDL, a signal named in relation to the type of error is set to 1, along with an error condition enable signal. These error enable signals allow the register process depicted in Fig. 4.7 to set the corresponding bit in the error_o vector. The error vector is only cleared by an entire system reset or being set by a control signal sent from the FIM. It was implemented in this way so that the error register in the AXI peripheral will hold its value until the FIM has acknowledged the error.

```
348
349     err_reg : process(clk, ns_proc_error_enable, err_cond_error_enable, sync_reset_n)
350     begin
351        if (sync_reset_n = '0') then
352            error_o <= (others => '0');
353        elsif(rising_edge(clk)) then
354            if(ns_proc_error_enable = '1' or err_cond_error_enable = '1') then
355                if(addr_align_err = '1') then
356                    error_o(0) <= '1';
357                else
358                    error_o(0) <= '0';
359                end if;
360
361                if(data_align_err = '1') then
362                    error_o(1) <= '1';
363                else
364                    error_o(1) <= '0';
365                end if;
366
```

**Figure 4.7** First portion of Error Register process (all signals encoded, only first 2 displayed)

A separate process (Fig. 4.8) is implemented to handle signaling the FIM when a process has occurred. If any of the error enable signals have been set, this process will create a pulse lasting four system clock cycles. The DCP FSM should continue to attempt to parse information from the SPI signals to meet the requirement that HECAD should be as passive as possible. Generally, the HRIM will not send information to the I2M while in an error state. The behavior of the HRIM will depend on the type of error detected. Specific details of this

```
1057
1058        err_state_cntr_rst <= '1';
1059        error_int <= '0';
1060        ns_error_int <= ps_error_int;
1061
1062        case ps_error_int is
1063
1064            when err_state =>
1065                err_state_cntr_rst <= '0';
1066                if(err_state_cntr < 4) then
1067                    error_int <= '1';
1068                else
1069                    error_int <= '0';
1070                    ns_error_int <= wait_high;
1071
1072                end if;
1073
1074            when wait_high =>
1075                err_state_cntr_rst <= '1';
1076                error_int <= '0';
1077                if(err_cond_error_enable = '1' or ns_proc_error_enable = '1') then
1078                        ns_error_int <= wait_high;
1079                    else
1080                        ns_error_int <= wait_low;
1081                    end if;
1082
1083            when wait_low =>
1084                 err_state_cntr_rst <= '1';
1085                 if(err_cond_error_enable = '1' or ns_proc_error_enable = '1') then
1086                    ns_error_int <= err_state;
1087                 else
1088                    ns_error_int <= ps_error_int;
1089                 end if;
1090
1091            when others =>
1092
1093        end case;
1094    end process error_interrupt_proc;
```

**Figure 4.8** Generating the error interrupt signal to the FIM

behavior will be presented in each error classes' description in the rest of this section. All of the problems within the transaction error class are detected inside the DCP FSM (4.6), while the timeout and configuration errors are set outside in separate processes.

### Transaction Errors

The first possible error that can occur during a transaction is the address alignment error. It is checked within the address super state (Fig. 4.9): if the CS signal comes back up high at any point during this state, the HRIM will assert the corresponding error signal. The DCP FSM will return to the idle state. This effectively skips this transaction, as no part of it is valid.

The next possible error is an invalid address. While not classified as a transaction error, it is detected in the address_check state (Fig. 4.10) inside the DCP. This state ensures

```
546             when addr_state_high =>
547                 if(err_cond_error_enable = '1') then
548                   next_state <= wait_low;
549                 else
550                     if(sync_sclk = '0') then
551                         next_bit <= present_bit + 1;
552                         sclk_period_rst <= '1';
553                         next_state <= addr_state_low;
554                     elsif(sync_ss_n = '1') then
555                         ns_proc_error_enable <= '1';
556                         addr_align_err <= '1';
557                         next_state <= wait_low;
558                     else
559                         next_bit <= present_bit;
560                         sclk_period_rst <= '0';
561                         next_state <= present_state;
562                     end if;
563                 end if;
```

**Figure 4.9** Detecting Address Alignment Problems in VHDL

```
592             when addr_check =>
593                 if(err_cond_error_enable = '1') then
594                   next_state <= wait_low;
595                 else
596                     if(to_integer(unsigned(addr_buff(6 downto 0))) > ADDR_VALID_HIGH) then
597                         addr_oob <= '1';
598                         ns_proc_error_enable <= '1';
599                         next_state <= wait_low;
600                     else
601                         next_state <= data_state_high;
602                     end if;
603                 end if;
604
```

**Figure 4.10** Detecting Valid Addressing in VHDL

that the address falls within segments of address space that Aries is configured to access. It too transitions the DCP back to the idle state to wait for the next transaction. The next error is the data alignment error. It is almost identical to the address alignment error, only varying in which portion of the transaction it occurs. Again, the CS signal is monitored to ensure that it is not pulled high prematurely. If such an error occurs, the HRIM is currently configured to toss out the entire transaction and return to the idle state. It could be argued that if any full registers have been accessed, this information should be passed to the I2M. This was not implemented however, as the presence of this error could indicate that the previous transaction was compromised.

The last transaction error detected in the DCP FSM is continuous-mode overflow (Fig.

```
607             when data_state_high =>
608             if(err_cond_error_enable = '1') then
609                   next_state <= wait_low;
610                else
611                   if(sync_sclk = '0') then
612                      if(present_bit= d_width) then
613                         if(present_data_length = MAX_BURST-1) then
614                            burst_overflow <= '1';
615                            ns_proc_error_enable <= '1';
616                            next_state <= wait_low;
617                         else
618                            next_bit <= 1;
619                            sclk_period_rst <= '1';
620                            next_data_length <= present_data_length + 1;
621                            next_state <= data_state_low;
622                         end if;
623                      else
624                         sclk_period_rst <= '1';
625                         next_bit <= present_bit + 1;
626                         next_state <= data_state_low;
627                      end if;
```

**Figure 4.11** Continuous-Mode Overflow

4.11). This check, also present in the data_high state, ensures that the number of bytes counted so far does not exceed the maximum length of the MOSI and MISO buffers. This value is set in the SPI HECAD VHDL package. For this design, the implemented buffers have 14 bytes of capacity. This is set based on the Aries IMU code. The FCS will never request more than the 14 bytes of data from the sensor data registers. So this buffer can reasonably set to this value, though the length is pulled out into the package in the event that future use dictates that this be longer. The check occurs when the same conditions are met to decide between terminating the transaction or extending the continuous-mode operation. If the number of bytes exceeds the 14, the DCP FSM will transition to the data latch state, but noting the error to the FIM by setting the corresponding error signal. The DCP FSM will naturally transition back to the idle state, where it will not capture erroneous data for the remainder of that SPI transaction.

### Signal Timeout Errors

The signal timeout errors, one error signal per SPI signal, all have their own state machines. Their functionality is virtually identical, only differing in which signal is monitored. For the sake of brevity, only the timeout monitoring for the SCLK signal will be presented.

It should be noted that there are three other sets of VHDL, differing only in the signal names used, present in the HRIM.

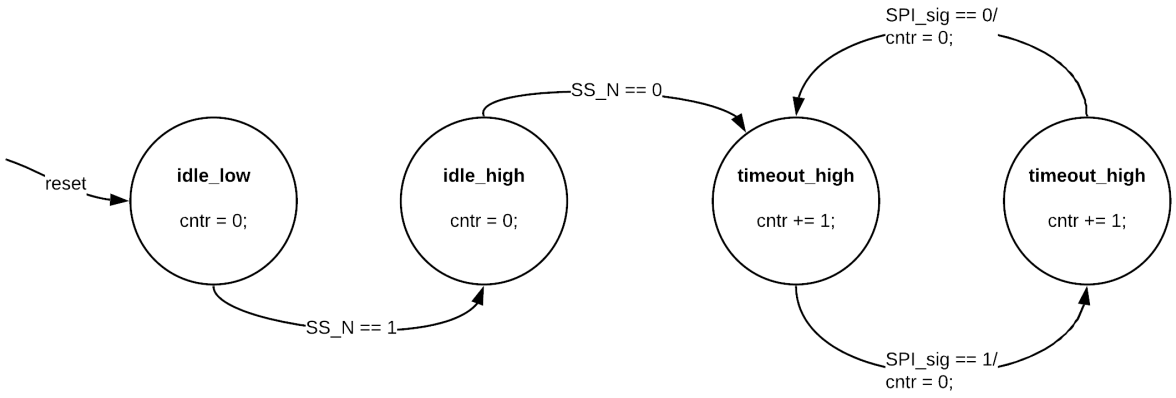


**Figure 4.12** Signal Timeout State Transition Diagram

The state transition diagram for a timeout FSM is displayed in Fig. 4.12. The initial states ensure that if the SPI HRIM is booted before Aries, the time spent where the system is idle does not trigger the timeout conditions. It follows the same logic and naming convention as the DCP FSM. The idle states wait for the CS signal to drop low, signifying that an SPI transaction has occurred. After this, it can always be guaranteed that regular bus activity will occur. For components not regularly sampled like the IMU, this functionality can be removed from the HRIM. Once the timeout FSMs are initiated, it enters another high/low superstate. The FSM transitions back and forth between the two substates whenever an edge is detected on the corresponding signal. When this occurs, a reset signal is pulsed for one clock cycle. The counters reset by these signals will count the number of system clock cycles between any activity on that signal. The actual timeout error signals are set in a separate process, responsible for both the timeouts and the mode configuration errors.

The condition for a timeout is determined by the value of these counters exceeding a threshold calculated based on generics and constants set in the SPI HECAD VHDL package. The constant "SPI_TIMEOUT_PERIOD" is calculated as:

$$= \frac{CLK\_FREQ}{SPI\_SAMPLING\_FREQ} * SPI\_TIMEOUT\_ROUNDS,$$

where the CLK_FREQ is the system clock frequency, the SPI_SAMPLING_FREQ is how many times the SPI device is sampled per second, and the SPI_TIMEOUT_ROUNDS is how many transactions the signal has been inactive. These are configurable constants so that the HRIM can be matched to whatever Aries hardware setup is present. There are situations where it is feasible that a signal will not change during normal sampling operation. This could occur if the maximum address is always read. If a read is encoded as '1', and the maximum address is also all '1's, then it is possible that the MOSI signal will never change from '1', as it typically idles at a high value when reading. This situation is unlikely, but modifications to the SPI HRIM would be required to handle this case. The other signals, as long as the peripheral is consistently sampled, will always be guaranteed to change. If the condition trips, the corresponding timeout error signal is set along with the error enable signal. This will cause the DCP to stay in the idle state until activity is detected again. It prevents incorrect data from being passed to the I2M.

**Mode Configuration Errors**

The mode configuration errors are set in the same process as the timeout error signals. No special FSMs or counters are required for these signals, as they can be detected by ensuring other portions of the design are not exhibiting a set of conditions. The conditions for a CPHA mismatch are as follows: the DCP FSM needs to be in the address or data superstates, it is not the first bit of either state, and either data (MOSI or MISO) signal's timeout counters are not a small value at the same time as a counter tracking the SCLK period.

Fig. 4.13 displays the two conditions as written in VHDL. This is essentially checking to make sure that neither device has just shifted a value at the same time as an SCLK edge that should have signified a sample. The SCLK period counter counts the system clock periods

```
965         --Master is mode 1, slave is mode 0
966         if((present_state = addr_state_low) or (present_state = addr_state_high) or (present_state = data_state_low)
967             or (present_state = data_state_high)) and (present_bit > 0)) then
968
969             if((sclk_period_cntr < 2) and  (mosi_timeout_cntr < 2)) then
970                 mosi_shift_err <= '1';
971                 err_cond_error_enable <= '1';
972             else
973                 mosi_shift_err <= '0';
974             end if;
975         else
976             mosi_shift_err <= '0';
977         end if;
978
979         --Master is mode 0, slave is mode 1
980         if((present_state = addr_state_low) or (present_state = addr_state_high) or (present_state = data_state_low)
981             or (present_state = data_state_high)) and (present_bit > 0)) then
982
983             if((sclk_period_cntr < 2) and  (miso_timeout_cntr < 2)) then
984                 miso_shift_err <= '1';
985                 err_cond_error_enable <= '1';
986             else
987                 miso_shift_err <= '0';
988             end if;
989         else
990             miso_shift_err <= '0';
991         end if;
```

**Figure 4.13** CPHA mismatch for both cases

for every whole period. This is why the SCLK timeout counter value cannot be used, as that always resets on every edge. A CPOL mismatch is checked by verifying that the SCLK signal is at the right value at the start of a transaction. For a device in mode 0, the SCLK signal needs to be low as soon as the SS signal drops. The violation of this condition will signify that the master is not configured with the correct CPOL value. The SCLK period counter is also used in the last error check, ensuring the SCLK is running at the right frequency. The SCLK frequency check ensures that the value of the SCLK period counter is within what is specified in the VHDL package. Any of these errors are considered fatal and will cause the HRIM to lock itself in the idle state.

## 4.6 I2M

### 4.6.1 Component Description

Like the HRIM, the I2M takes in the system clock and reset signals, which drive a much simpler FSM than the DCP present in the HRIM. The I2M takes in the four vectors of data created by the HRIM. It does this whenever the data_ready signal is sent from the HRIM. The I2M has ports that pass this information through to the AXI peripheral, where it can be retrieved by the FIM. The FIM is signaled with an interrupt from the I2M whenever new

valid data has just been written to the AXI registers. In addition to the data signals, the I2M has its own error vector and error interrupt signal just like the HRIM. These interfaces are maintained to be consistent across the two submodules. It aids in readability and makes the top level module that combines the I2M and HRIM easier to implement.

### 4.6.2 I2M Finite State Machine

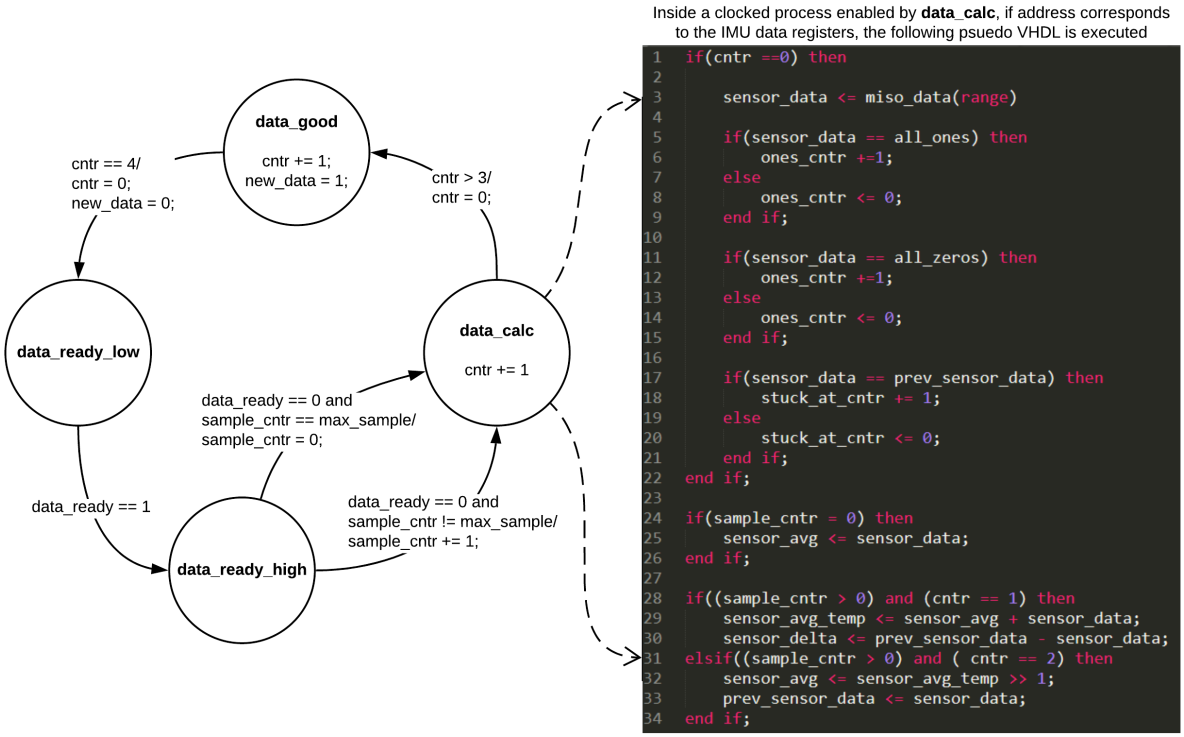


**Figure 4.14** Main FSM inside I2M

The main FSM present in the I2M only has four states. Fig. 4.14 displays the state transition diagram for the I2M FSM. The initial state is named data_ready_low. This naming scheme follows in the same manner as the high/low convention adopted in the HRIM. Therefore, the data_ready_low state is primarily waiting for the data_ready signal to go high, set by the HRIM. Once it does, the FSM transitions to the data_ready_high state.

This state is used to capture the data present on the input ports of the I2M into its own registers. The state serves as an enable signal for the clocked process to create a register. Additionally, the data_ready_high state is used to increment a counter that keeps track of how many samples have occurred. The sample counter is used in calculations related to the error checking.

After the data_ready signal drops back low, the FSM transitions to a state named data_calc . This state is used as an enable to a register process that calculates all of the required information to verify the validity of the data coming from the IMU. If the address of the transaction matches the base for the sensor registers, the pseudo VHDL displayed in Fig. 4.14 will be executed. Note that the signal "sensor_data" is used to compress the VHDL present in this process, it represents all seven sensor readings.

On the first system clock cycle within this state (line 1), the MISO data is broken up into the individual sensor readings. The I2M checks for three separate conditions: if the sensor data is all ones (line 5), all zeros (line 11), or is equal to the previous sample (line 17). If any of these are true, a corresponding counter increments. If not, the counter resets to zero. The next clock cycle (line 28), the first step in calculating a running average is performed. The current average and the new sensor data are added together. In the same cycle, the delta value between the previous and current sensor data is calculated. The next cycle (line 31), the temporary average value is divided in half to calculate the new average. Finally, the previous sensor value is updated for the next sample. Note that the average value is reset periodically depending on the sample counter (line 24).

The next state really only serves to set the interrupt pulse length to the FIM. This state also waits for four clock cycles. The interrupt will be set to high for the entirety of the FSM's duration in this state. After four clock cycles the FSM will transition back to the initial state to wait for further transactions.

### 4.6.3 Error Checking

The remainder of the I2M performs the error detection outlined in Section 3.3. Some additional processes are required to capture the remaining functionality required by the I2M. The first is effectively identical to the output error register present in the HRIM, displayed in Fig. 4.7. It takes the error signal names and encodes them in one-hot fashion to a single vector that is output to the AXI register. The second is essentially a clocked register process. It calculates various values based on the sensor data sent up from the HRIM.

The pseudo VHDL in Fig. 4.14 is present inside the "else" case of the synchronous reset condition. This makes all of these calculations done on the rising edge of the system clock, with enable conditions, i.e. registers with enables. The checks performed in the I2M only consider the fourteen data registers that hold the sensor values. Any other type of transaction, specifically the ones used in the initialization, are captured and passed to the FIM, but no other type of information checks are performed. The condition to ensure the type of transaction is correct to run this data calculation is based on the address and the state of the I2M FSM. As long as the first value was a read, i.e. the most significant bit of the address signal is '1', and the rest of the address is 0xBB, which corresponds to the lowest sensor data register, and the present state of the FSM is data_calc, the MISO data will be segmented into separate registers. The gyro and accelerometer each have a register for their three axes, while the temperature sensor gets its own. While fourteen registers are read because of each sensor and axis having a "high" and "low" register on the IMU, this process combines them into a single 16-bit value. This is the only way to easily perform math functions on the data.

There are many values maintained by the I2M based on the sensor data, used in determining whether a potential fault has occurred. Note, each of the sensors have their own type of register discussed further but again are referred to as a collective. This condition uses the sample counter and the system clock. The first block of calculations finds a sliding average

and delta of all the sensor values. This calculation only occurs across the first two system clock cycles after one sample has already been captured. This is because the sliding average is calculated by adding the new sensor data values to the running average and then dividing by two. In this case, the VHDL implementation uses the shift_right function in the IEEE arithmetic library. The shift_right function automatically detects the signal type, which in this case is signed, and will perform arithmetic shift right (as opposed to a logical shift if it were unsigned). Shifting right by one bit position is equivalent to dividing by two, so the average is maintained. The delta values are found by subtracting the previous data from the new data. The previous data is captured after these calculations finish.

The last process sets the error signals if any of their conditions are met. The discontinuity check ensures that delta values do not exceed a certain threshold. Note, all threshold constants are set in the VHDL package. These values are experimentally derived and differ for each sensor. A heuristic was adopted for initially setting the threshold values, as this system has not been flight tested. This heuristic sets the threshold at any jump between two samples that are larger than a fourth of the total range of the sensor. For example, the accelerometer is configured to have a range of +/- 4 G's. So any difference between consecutive samples larger than a fourth of this range would constitute a discontinuity. It is worth noting that because both the gyro and accelerometer measure rates and acceleration, respectively, momentary spikes in values can be expected. The exact degree of what is or is not a valid pulse has been left as future work for this system.

The next set of checks performed ensure the values of the sensor registers are not approaching the extrema of the configured range. Again, a heuristic is used to set these threshold values. For a range check, any value within 5% of the maximum and minimum value supported will be reported as a violation of this condition. For all three of the sensors, these thresholds are representative of extreme physical phenomena, so it is not as uncertain that this correlates to a serious problem as with the discontinuity checks. It should be noted that these values are checked against the sliding average, so these conditions must be sus-

tained for a significant amount of time. This class of detection is more about ensuring the sensor value is not clipping, though this is not as important for these sensor values given how disastrous any operation anywhere close to these values is.

Similar to the range checks are the reasonability checks. The threshold values for the acceleration and orientation have been set to 10% of the maximum and minimum values. This is still a heuristic but is based on what the airframe could reasonably withstand. These values correspond to +/- 3.6 G's and +/-900 degrees per second, respectively. Again, having a sensor value average to above these ranges will still represent very unlikely correct operational behavior. The temperature is set based on another metric. Since the value represents the temperature of the device, the reasonability thresholds are set to the range listed in the specification sheet for the MPU9250 before the other sensor values become unreliable. The curves of the other two sensors are modified depending on the temperature, so any reading outside this range is cause for alarm. Anything outside this range is unreasonable for extended operation of an integrated circuit.

Finally, the stuck-at error is detected. The values of counters set in Fig. 4.14 are checked against thresholds. The ones and zeros counters are compared against a much smaller threshold than the stuck_at threshold. This is because these sensors will rarely hold these values for more than two or three samples and can indicated either a sensor problem or a physical fault in the SPI communication. Stuck_at thresholds are harder to create a heuristic for without available flight testing. It can reasonably be assumed that the vibrational noise on-board a UAS will always cause the acceleration and orientation values to change almost every cycle. These thresholds have been set to 50 samples. The temperature sensor again requires separate consideration. It is still likely to fluctuate while operating correctly but is not as susceptible to noise as the other two sensors. It has been set to trigger when 200 consecutive samples have returned the exact same value. It is likely that after further experimentation and flight testing, these values can be reduced. They have been set as a baseline that can be further refined.

# Chapter Five

# Results

## 5.1 Testing Setup

A testing setup is needed to verify that the design is working as intended. Traditional methods of verifying FPGA designs typically start with simulations performed with testbench components wrapped around components under test. This can serve as a baseline of design functionality but simulation has serious limitations that have affected the way SPI HECAD has been verified. The first issue with simulation is the complexity of providing stimulus at this level. Starting with even just the sensor data capture portions, SPI HECAD needs a functioning SPI bus to monitor to correctly exercise the state machines in both modules. The HRIM in particular is sensitive to this, as it will expect certain aspects of the bus to always hold true. The implementation of SPI protocols can differ from master and slave devices, but original testing and simulation on only the DCP FSM were performed using DigiKey SPI IP cores. The IP cores are implemented in VHDL, so they were instantiated inside a testbench where SPI HECAD could take in the SPI signals as its designed to. These cores run in SPI mode, however, but modifications were made to allow SPI HECAD to correctly interpret the bus. This provided a false sense of security, as initial results displayed correct functionality. Further development on the I2M also showed the module operating correctly.

The usefulness of simulation hit its end when the error detection for both the HRIM and

I2M required verification. Neither DigiKey IP core provided an easily extendable design so that modifications could be made to make them more resemble Aries and the IMU. The need for real IMU data is heavily stressed, as many of the heuristics for detection thresholds are based on these experimental results. Writing simulation models to emulate Aries and the IMU seemed to cause too much design complexity when other solutions exist.

The final issue that proved simulation to be a too cumbersome and inaccurate form of design verification is differences in the design when implemented in hardware. Initial forms of verifying the design after being implemented in the FPGA (as opposed to simulating) displayed serious issues. These were predominantly caused by the external nature of the inputs. SPI signals coming from outside the FPGA are not synchronized with the system clock like the signals provided by IP cores simulated alongside the design. This issue, along with other differences in behavior between simulation and hardware implementation caused simulation to be abandoned as a form of verification altogether. As such, only results from the testing system described in the following subsections will be presented, not from these preliminary simulations.

### 5.1.1 Hardware

The SPI HECAD design is tested on the Avnet UltraZed, the same board the whole HECAD system targets. As mentioned, testing is done in a way that exercises the implemented FPGA design. The VHDL is run through Xilinx's Vivado tool which processes the text files into information used to program the FPGA. An Aries Tiny FCS was modified for the purposes of testing SPI HECAD, electrical contacts were placed on the actual SPI bus. These leads can then be fed directly into the FPGA via a PMOD connector. This slightly modifies the design from its eventual final form. In the combined HECAD design, all hardware bus signals on the FCS will be electrically isolated by the FPGA. HECAD will have to connect these signals internally as it sits directly between all of them.

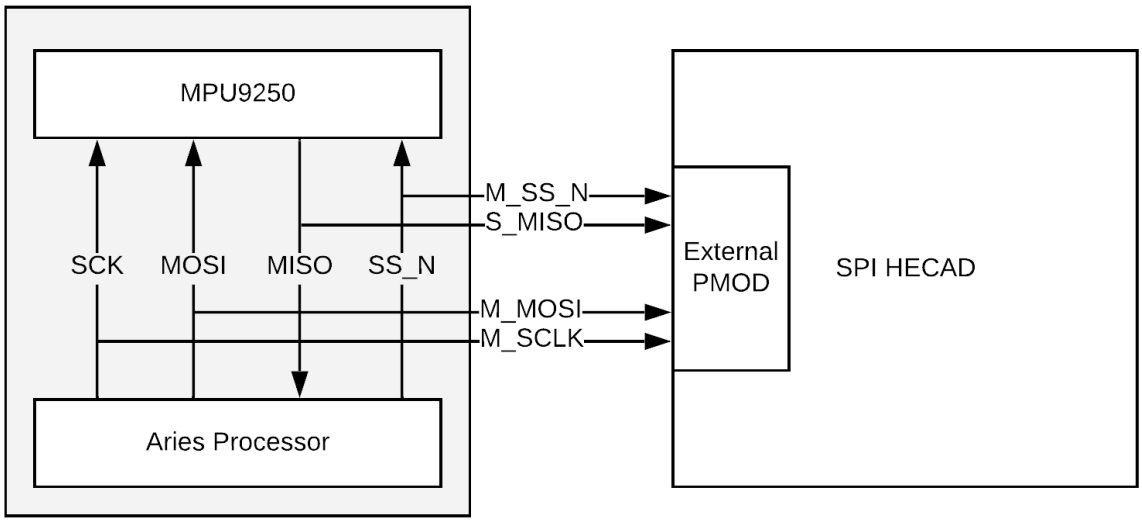


**Figure 5.1** Testing Hardware Block Diagram

The testing hardware setup, presented in Fig. 5.1, deviates from this requirement, though not in a significant way in terms of functionality. The connections to the SPI bus represent soldered leads on the Aries board, they do not break the connections between the FCS processor and the IMU. This does not create any problems for the robustness of testing in this manner, as the HRIM only cares about the logical values of these signals. Error mitigation or handling is not within the scope of this thesis, so the SPI HECAD modules do not need the ability to intercept or otherwise modify the signals between the Aries components. This is the reason that no outputs of SPI HECAD are present in that block diagram, as they are unused for this testing scheme.

### 5.1.2 Xilinx Integrated Logic Analyser

The Xilinx Integrated Logic Analyser (ILA) is an extremely useful tool for monitoring FPGA designs in real time. Probes are attached internally in the FPGA fabric to signals requested by the user. The values of these signals can be sampled over a period of time. It provides

real signal information while the system is running. Although typically aimed at debugging, the ILA system still provides a very attractive option for verifying design functionality. This is especially true given that it is monitoring the SPI HECAD system in its most accurate form. SPI HECAD is receiving real SPI signals from the device it is built to monitor and the ILA can provide the actual values of internal signals as the system processes transactions.

There are limitations to the ILA but nothing that degrades the accuracy of the results. The sample depth, or the amount of time that can be captured, is a function of how many probes are attached to internal signals. As the ILA requires FPGA fabric, the hardware resources required increase with the number of monitored signals and the sample depth. This creates an upper bound on how many samples can be captured at one time. Fortunately, for this design, the entire fourteen byte transaction can be captured with sufficient signals monitored. Other adverse behavior has been observed and believed to have been caused by the ILA's interference. It can cause timing closure problems if a design is already close to failing slack requirements. Despite these limitations, no accuracy in the testing results is sacrificed when verifying the design in this way. For these reasons, the ILA will be used as the primary source of verification results. All of the waveforms presented in the remainder of this chapter will be captured from ILA results.

## 5.2 SPI Fault Injector

### 5.2.1 Overview

A VHDL component was written to inject faults into the SPI HECAD system. A block diagram that presents how the SPI Fault Injector (SFI) fits into the overall SPI HECAD design is present in Fig. 5.2. The fault injector sits in between the signals coming from outside the FPGA and the SPI HECAD system. It intercepts these signals and modifies them in a controllable way. The SFI is implemented as another AXI peripheral.

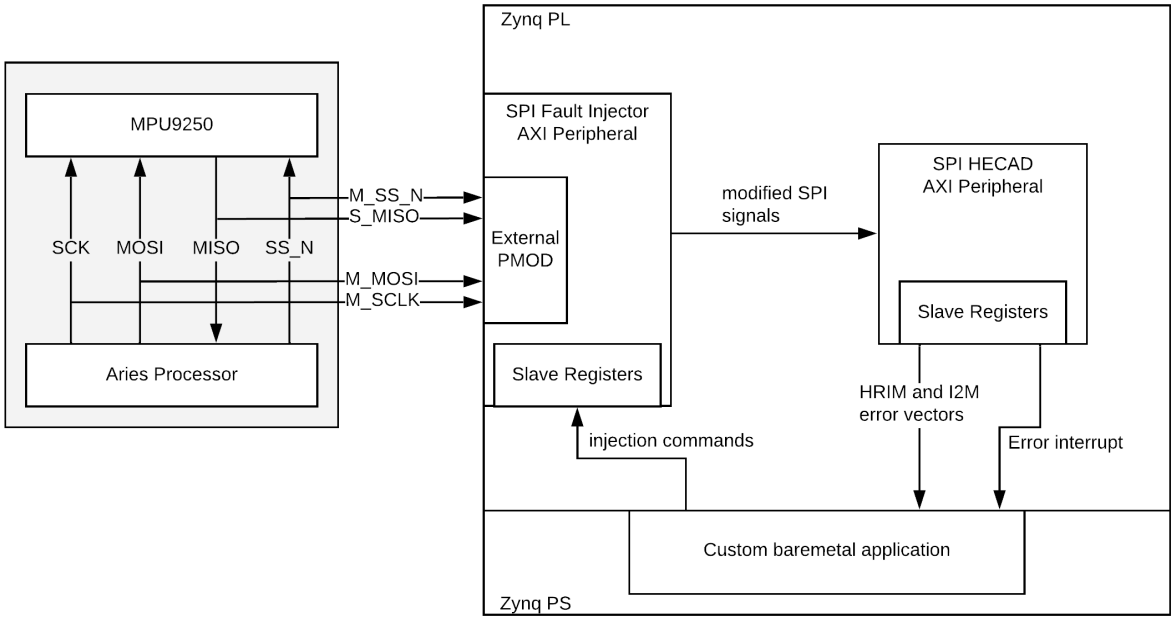


**Figure 5.2** SPI Fault Injector Block Diagram

The implementation of the SFI is heavily based around the SPI HRIM. It is a stripped down version of the DCP FSM used to drive the sensor data capture inside the HRIM. The transaction error detection functionality is removed, but otherwise the state machine is left intact. Various timers, registers, and processes are copied over from the HRIM where required. The biggest difference between the HRIM and the SFI is how the SFI treads the outbound signals. Running in the default mode, the SFI does not modify the signals in any way and lets them pass through unperturbed. The waveforms in the following subsections will all be pulled from this design. Results from SPI HECAD operating under normal conditions will still have the SFI component hooked on the front of the design, though its presence has no effect on the functionality of the design.

As mentioned, the SFI is a separate AXI peripheral. This has many benefits. The first and foremost positive is that it allows controlled execution of different injected attacks during runtime. A corresponding PS application is developed to write values to the SFI

AXI registers. The values of these slave registers are passed down directly into the SFI component. It takes this vector and interprets each bit as a separate attack to be performed, encoded in a similar manner as the error reporting performed by the HRIM and I2M. The VHDL implementation for these attacks and emulated faults will be presented alongside the results in the following sections.

The primary method of fault inject is performed by modifying the SPI signals outbound to the SPI HECAD system. The SFI does not have access to any of the internal components of SPI HECAD; it can only assert faulty values on the four SPI signals. This means, for example, that the information errors will be exercised by flipping specific bits in the fourteen byte transaction. Performing attacks in this way verifies SPI HECAD to a much higher degree than providing access to the design for the SFI. Modifications of the bus signals must first propagate through the HRIM then onto the I2M. This ensures that the SFI is not introducing any variations in the way SPI HECAD functions normally, stressing only the error detection portions.

## 5.3 HRIM Fault Injection

This section and Section 5.4 will present the results of the SPI HECAD subsystems when attacked by the SPI Fault Injector. As previously mentioned, the corresponding implementation of the SFI for the specified attack will be presented before each of the results. The waveforms will be compared against how the system behaves without the presence of fault injection. Note that all of these waveforms are captured from the SPI HECAD hardware interacting with a real Aries FCS during runtime and are exactly indicative of how the system will perform in these environments.

### 5.3.1 Base HRIM Functionality

The base functionality of the HRIM, primarily the DCP FSM, must first be presented in order to understand the results of the injected attacks. The way it parses the primary transaction of fourteen bytes of sensor data will be the framework upon which all of the results in this chapter will be presented against.

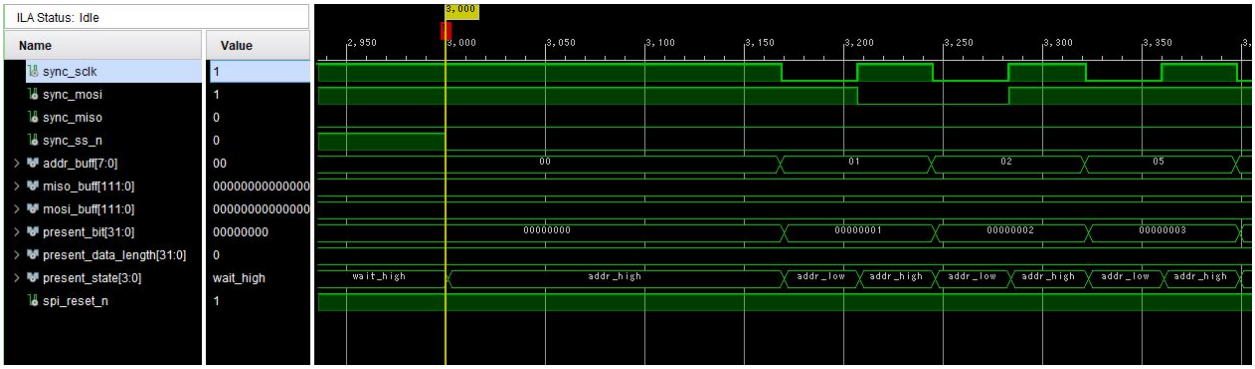


**Figure 5.3** First Half of DCP Address State

Fig. 5.3 displays the first portion of the sensor data transaction and will be used to explain how the remainder of the waveforms in this chapter will be formatted. The top four lines are the synchronized SPI signals: SCLK, MOSI, MISO, and the SS (appended with "_n" to denote active low) in that order. Next is the present variable for the DCP FSM. This is the best way to discern where in the transaction the DCP FSM is. The next three signals hold the buffered information parsed off the SPI bus, those being the address and all MOSI and MISO data, respectively. The following two counters, present_bit and present_data_length, keep track of the number of bits read within each byte and the number of bytes captured within the transaction, respectively. These values, combined with the state variable depict the current position of the window within the transaction. This will be important, as only snippets of the transaction will be presented at a time and in various scopes. Fig. 5.3 is zoomed in so signal values can be seen but other figures will display more of the transaction. At least this information will be provided in every figure and can be expected in this order

at the top of the waveform. More signals will be added to the bottom and referenced when displaying their corresponding functionality within the HRIM.

Now referring back to Fig. 5.3, the start of the SPI transaction is signified by the ss_n signal dropping low. The DCP FSM transitions to the addr_high state, waiting for the first SCLK edge, which occurs at 3,160 time units (TUs). The ILA treats a single TU as the period of the clock fed to the debug cores. For this system, this is a 50MHz system clock, making each TU 20 nanoseconds. They will be referred to in terms of TUs, however, as this value is present in the waveforms. Now the FSM will transition back and forth between the addr_high and addr_low states on every clock edge. The addr_buff begins to shift in the information present on the MISO signal on every falling edge of the SCLK. The bit counter can also be seen incremented for every address bit that is sampled. Note that the SCLK signal is inverted from what is being provided by the FCS because this allows the HRIM to be configurable for different SPI modes. These components are running in mode 0, so the SCLK signal would be inverted from its values in 5.3 and all waveforms presented.

Fig. 5.4 displays the last four bits in the address state (top) and a zoomed out view of the entire address state. Starting with the top, the bits continue to shift into the address buffer. When the SCLK signal returns high for the final time in this window at 3740 TUs, there is a small change in the present state. This, although too small to be displayed, is the DCP FSM entering the address_check state, where it will verify that the transaction is valid so far. If there was a problem with the address, which for this transaction can be seen in the bottom half as 0xBB, the HRIM would throw an error and return to the idle state, ignoring the rest of the invalid transaction. The address of 0xBB corresponds to the most significant bit signifying a read operation, which is encoded as a one. The rest of the address when this bit is taken off (0x3B) is the base address for the sensor registers. The state machine pauses for a brief time, as the SCLK signal stops pulsing. This allows the IMU to have time to bring in the requested data, ready to start shifting out on the next clock edge. With the state machine now in the data superstate (data_high and data_low) it is ready to continue


**Figure 5.4** DCP Address State End (Top), Full Address State (Bottom)

shifting in this information. Note that the DCP FSM does not consider the address as a part of the data length, this does not increment until after the first data byte has been read.

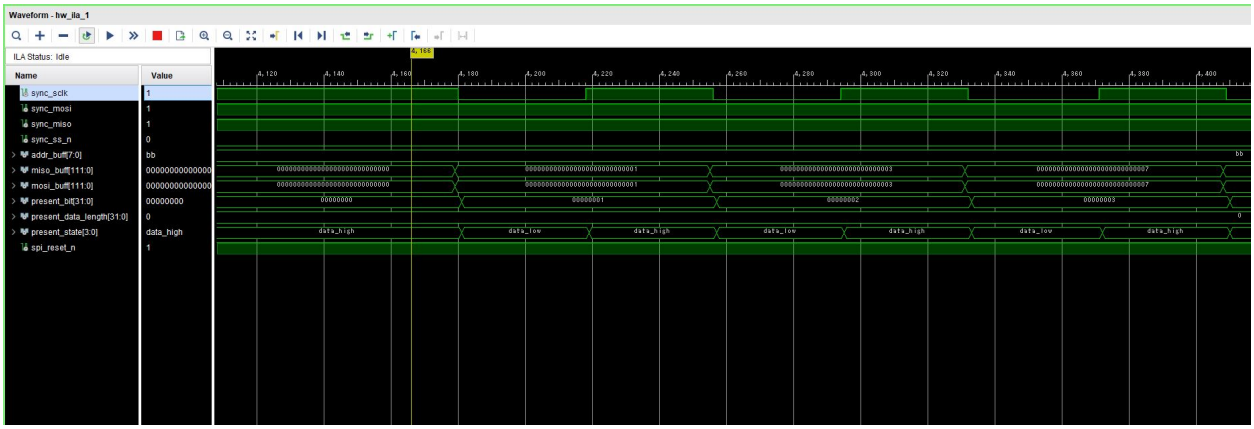


**Figure 5.5** DCP Data State

The very beginning of the data superstate can be viewed in Fig. 5.5. This waveform

is zoomed in so that the content of the MOSI and MISO buffers can be seen. As with the address, these buffers have whatever data is present on the corresponding signals shifted in and captured on the falling edge of the SCLK. The states transition back and forth between the high and low data states. The present_bit signal increments every time the data is shifted in, just like the address state. Unlike the address state, however, the data state will continue past reading in one byte. The next waveform, Fig. 5.6, displays the last few bits of the first byte and how the DCP FSM handles continuous mode operation.

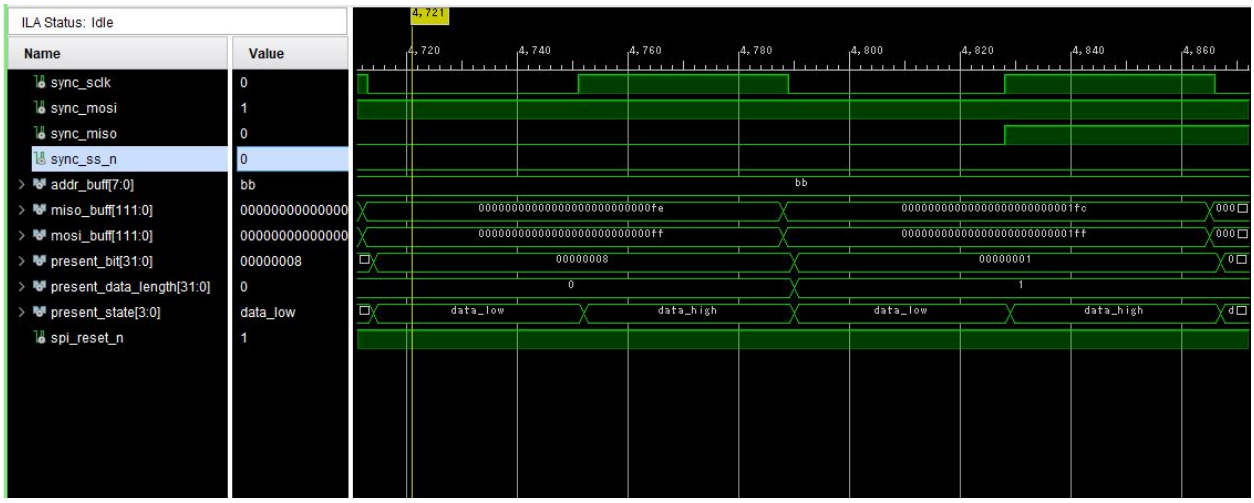


**Figure 5.6** One Full Byte of DCP Data State

The important concept to grasp from Fig. 5.6 is that the bit numbering within a byte is implemented in an initially counterintuitive manner. Typical numbering starts at zero and counts up to the length, minus one. As displayed, the DCP FSM is shown to count all the way up to eight, and roll back to one instead. The reason for this is how continuous-mode operation must be detected. After a byte is transferred, the transaction can end by the SCLK stopping and the SS_N signal coming back high. If the transaction continues like in Fig. 5.6, the SCLK will continue to run. It is not until this falling edge at around 4,785 TUs that the FSM can definitively determine that the transaction is not finished. The problem is that the bit counter has now spent a cycle at the value of eight, which is a required condition to differentiate between the start of a byte (i.e. the value cannot immediately roll back to zero).

The bit counter instead resets to one, as the previous bit was actually the first bit of the new byte. This only affects the way the transaction looks when viewed in a waveform. The actual information stored in the MOSI and MISO buffers are not affected by this numbering system. This process will repeat itself until all fourteen bytes have been read.

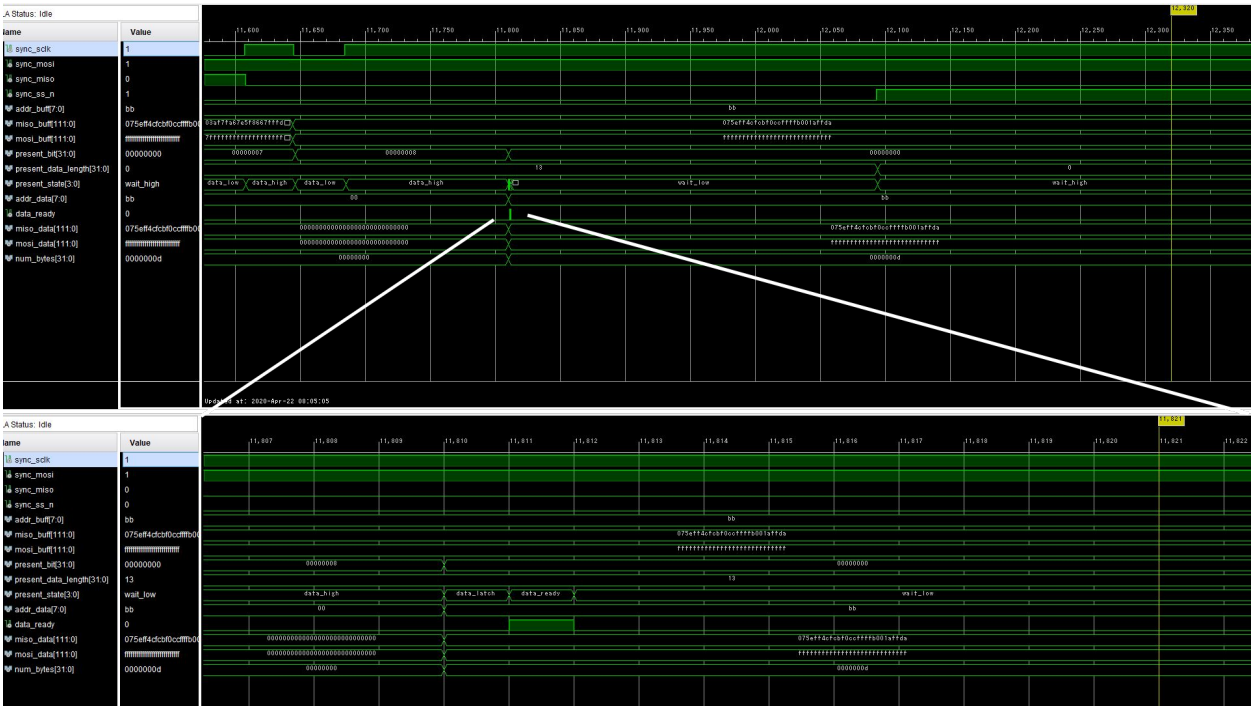

**Figure 5.7** End of SPI Transaction

Fig. 5.7 displays the end of a transaction. The top waveform shows the full version, while the bottom shows a zoomed in area denoted by the white bars. The additional signals display registers that hold the values of the buffers to send to the I2M. The data ready signal is also visible, which will signal the I2M that it has new information to process. After the last bit of the last byte is read, the SCLK signal stops and remains high. The DCP FSM detects this after well over an SCLK period has passed, meaning the transaction is done. It then cycles through the two states responsible for latching in the data and setting the data_ready signal which can be seen in the bottom waveform. After this, the state machine goes back to the idle state, where it will wait for the next transaction. By setting the DCP

FSM state back to idle, the buffers are cleared and the data_latch state will not be reached. This ensures no corrupted or malicious data reaches the I2M. When Aries is running in its normal mode, this cycle will continuously happen for the duration of its runtime.

### 5.3.2 HRIM Injection Results

This section will present how the SFI emulates attacks and faults, and how SPI behaves in response. Inside the SFI, a VHDL process takes in a vector of one hot encoded signals. This vector is set by an AXI register, which can be configured by a test PS program, where a user can select the type of attack performed in real time. The error_in signal displayed in the remainder of the waveforms is this register.

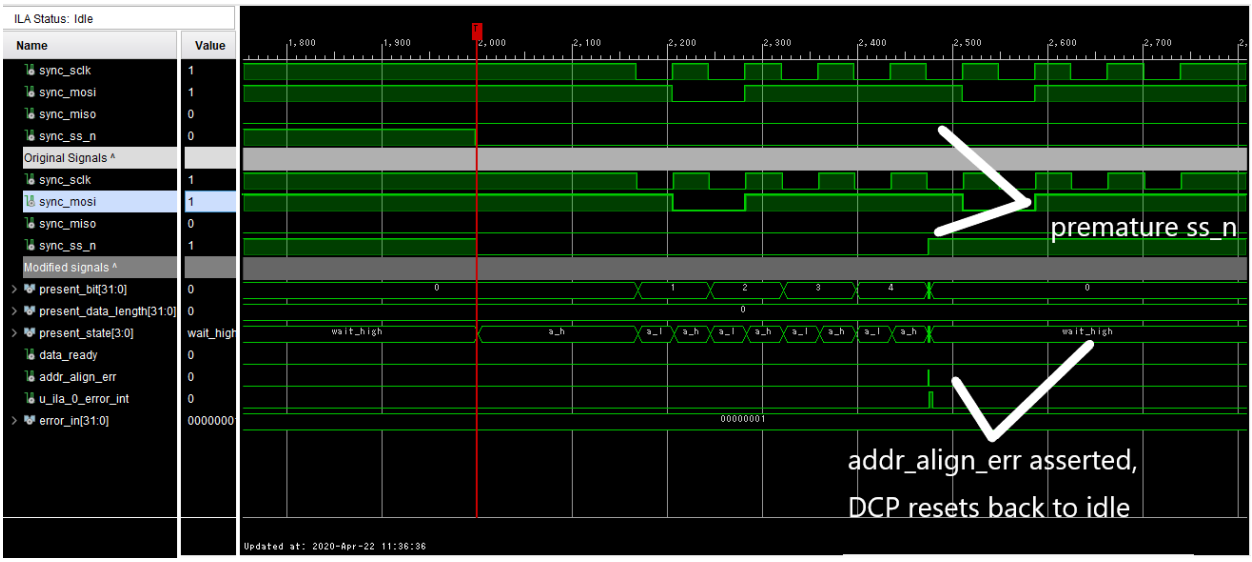


**Figure 5.8** Address Alignment Fault Injection

Beginning with the address alignment error in Fig. 5.8, this error_in signal can be seen at the very bottom of the displayed signals. Note that the DCP states have been abbreviated for brevity (addr_high is now a_h, addr_low is now a_l, etc). The original SPI signals are displayed at the top of the waveform, unmodified by the SFI. Below the divider, the modified signals are listed. Annotations on the waveforms themselves will denote where the

two groups of signals differ, exemplified in Fig. 5.8. For this attack, the SFI will set the SS_N signal to high in all cases except for the first few periods of the address state. After the fifth period, this signal is taken by the SFI and pulled high for the remainder of the transaction. SPI HECAD identifies this premature termination of the transaction and sets the addr_align_err signal. The error_int also pulses to notify the FIM of the incorrect bus behavior.


**Figure 5.9** Invalid Address Fault Injection

Fig. 5.9 displays the SFI writing an invalid address to the SPI HRIM. This is done by pulling the MOSI signal high to write 0xFE instead of the typical 0xBB. 0xFE corresponds to an unused peripheral register so Aries should never be accessing this address. The HRIM detects this invalid address in the addr_check state of the DCP FSM. When the address is deemed invalid, the transaction is stopped by setting the DCP back to the idle state. The corresponding error signal is set to alarm the FIM of the malicious tampering.

Fig. 5.10 displays the SFI prematurely pulling the SS_N signal high before the transaction completes. Only two bytes had been written before the attack cut the transaction short. The corresponding error signal pulses along with the interrupt to signal the FIM. The

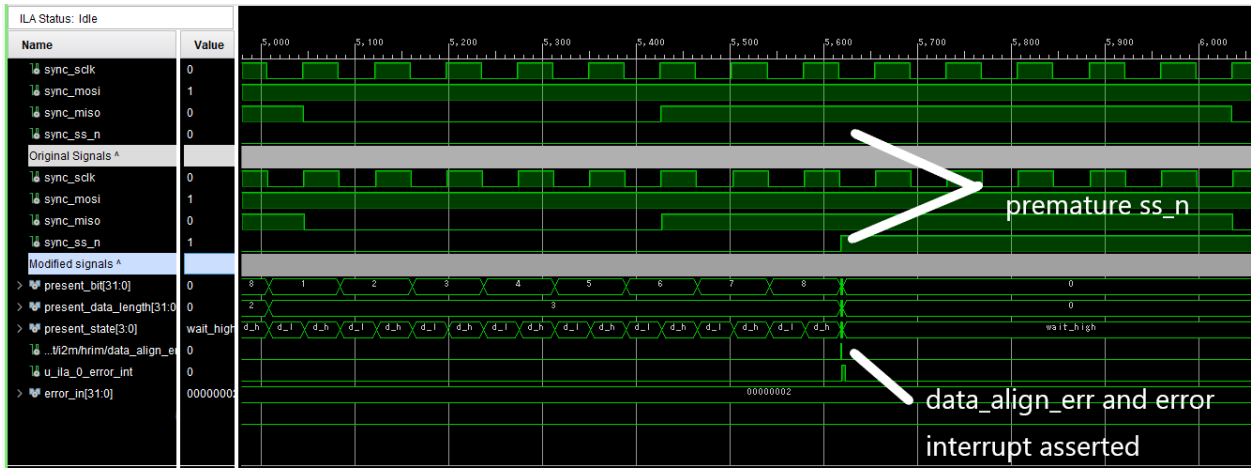


**Figure 5.10** Data Alignment Fault Injection

DCP FSM returns to the idle state ensuring the corrupt data is not passed on to the I2M.

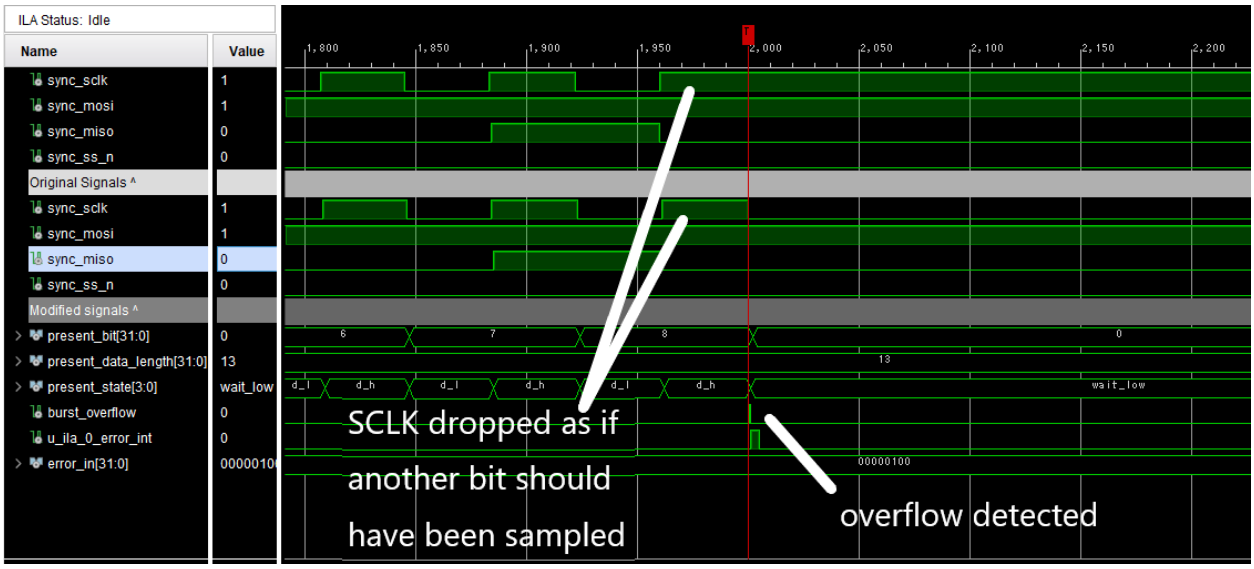


**Figure 5.11** MOSI/MISO Buffer Overflow Fault Injection

The MOSI and MISO buffers are of finite length, so if any SPI transaction attempts to run past the hard cap of fourteen bytes, these will shift out the relevant senor data. Fig. 5.11 displays the SFI attempting to perform this attack by dropping the SCLK signal after the fourteen bytes have already been sampled. This would indicate more information coming across the MISO signal. The SPI HRIM detects that this case should not happen and reports

the error signal and asserts the interrupt. The information from this transaction is thrown out, as Aries should never request more data than the fourteen bytes for the sensor data.

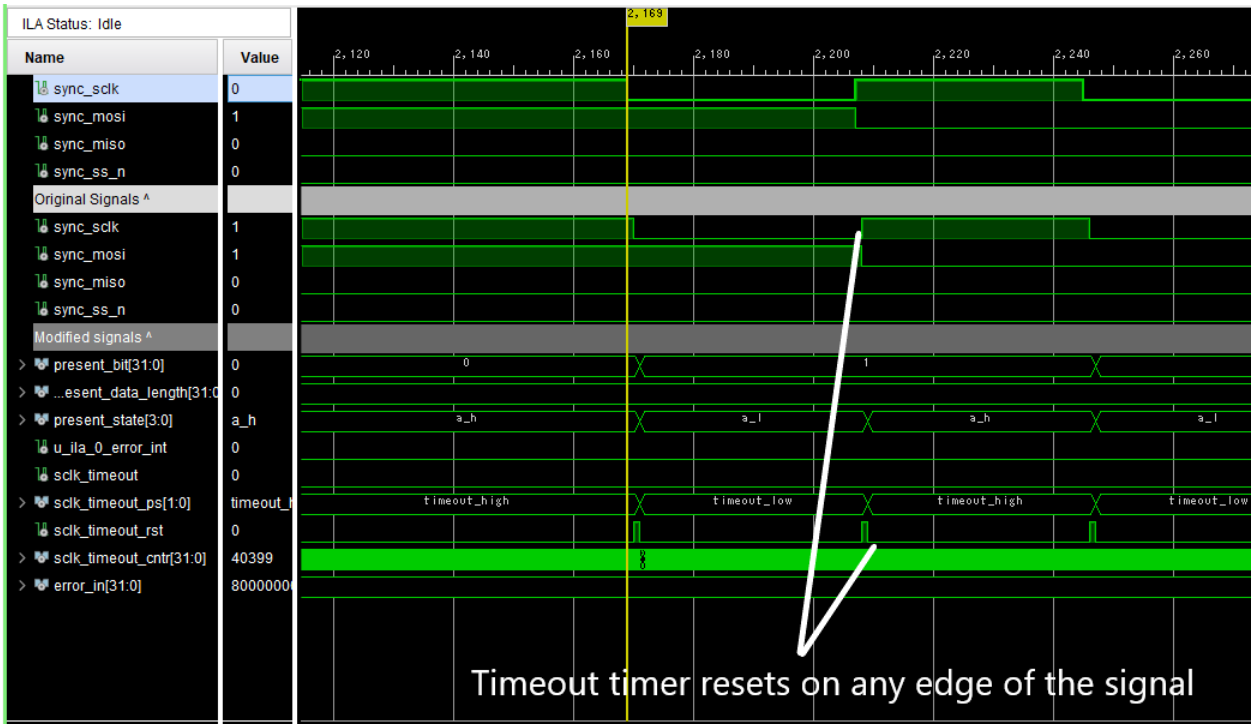


**Figure 5.12** Checking for Signal Timeouts

The next type of attack targets all of the SPI signals. The SFI is capable of cutting any of the four signals, exercising the timeout detection systems inside SPI HECAD. Depicted in Fig. 5.12 is the normal operation of the timeout mechanisms, the attack has not been performed yet in this waveform. This figure displays how internal timers keep track every time a signal changes. Only the SCLK timeout detection is displayed in these waveforms, as it works identically for all SPI signals.

Fig. 5.13 displays an entire transaction once the timeout attack has begun. The SCLK signal does not change and as a result the DCP FSM does not cycle through states. The timeout detection does not occur until a later time, as currently the threshold is set larger than one sampling period. Fig. 5.14 displays the sclk_timeout error signal finally firing once the threshold is reached.

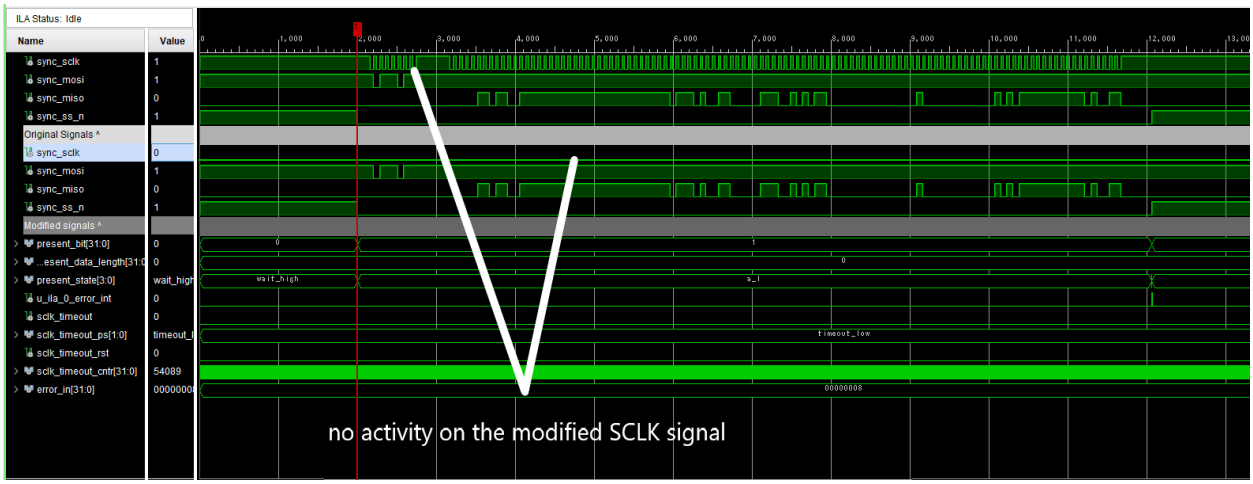


**Figure 5.13** SCLK Timeout Transaction

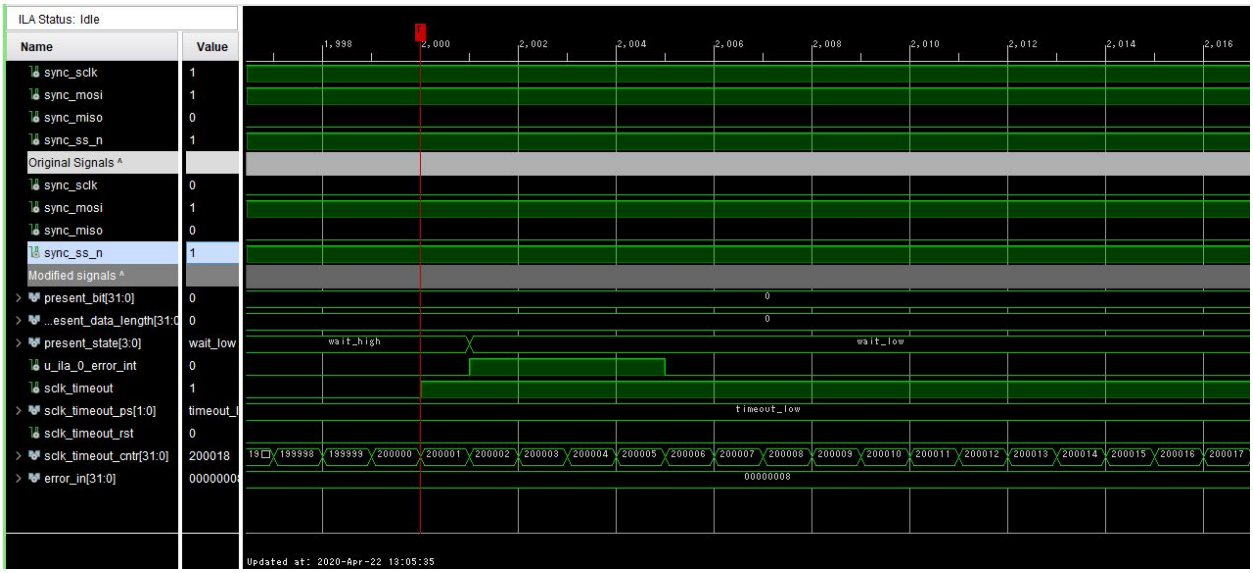


**Figure 5.14** SCLK Timeout Error Signal Triggered

As mentioned, the mechanisms for timeout detection for all signals work a similar manner, for this reason only the results for the SCLK timeout attack are presented. Note that the SFI and SPI HECAD are capable of performing and detecting attacks on all of the SPI signals.

Fig. 5.15 displays the SFI modifying the SPI signals to emulate a mode configuration error. This specific variant displayed above is creating a SCLK phase mismatch (mode 0 vs mode 1) by inverting the SCLK one half period after the transaction starts. This ensures

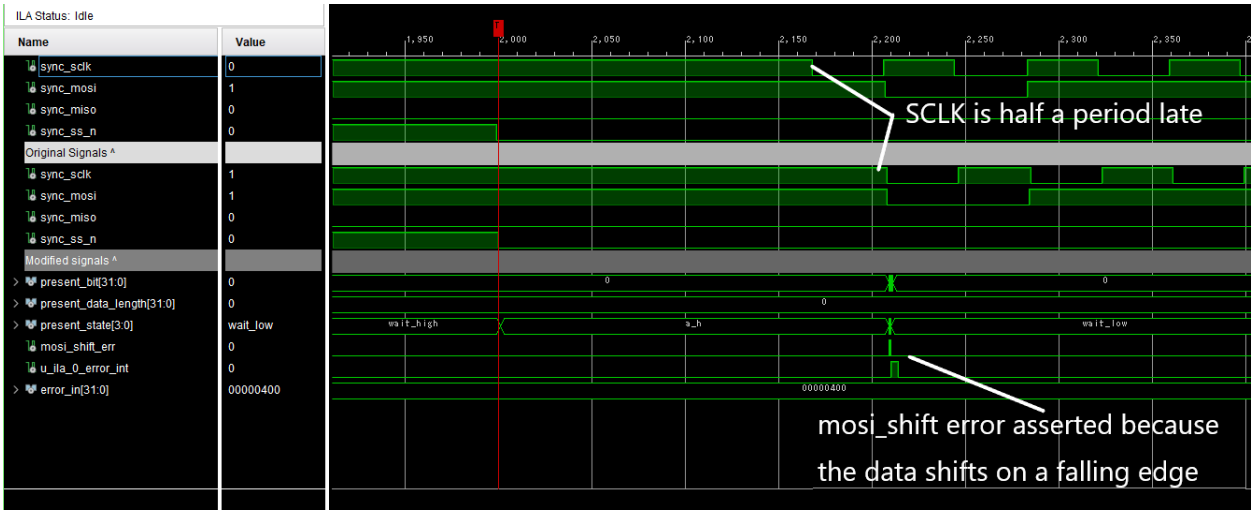


**Figure 5.15** CPHA Mode Fault Injection

that the SCLK frequency error is not tripped and keeps the functionality separate from a SCLK polarity mismatch. The erroneous SCLK edge occurs near a MOSI bit shift, causing SPI HECAD to detect the error and assert the corresponding error signal along with the interrupt. This is a fatal error and as long as it keeps occurring the DCP FSM will never leave the initial states. Although not presented, SPI HECAD is capable of detecting incorrect shifting/sampling behavior on the MISO signal as well.

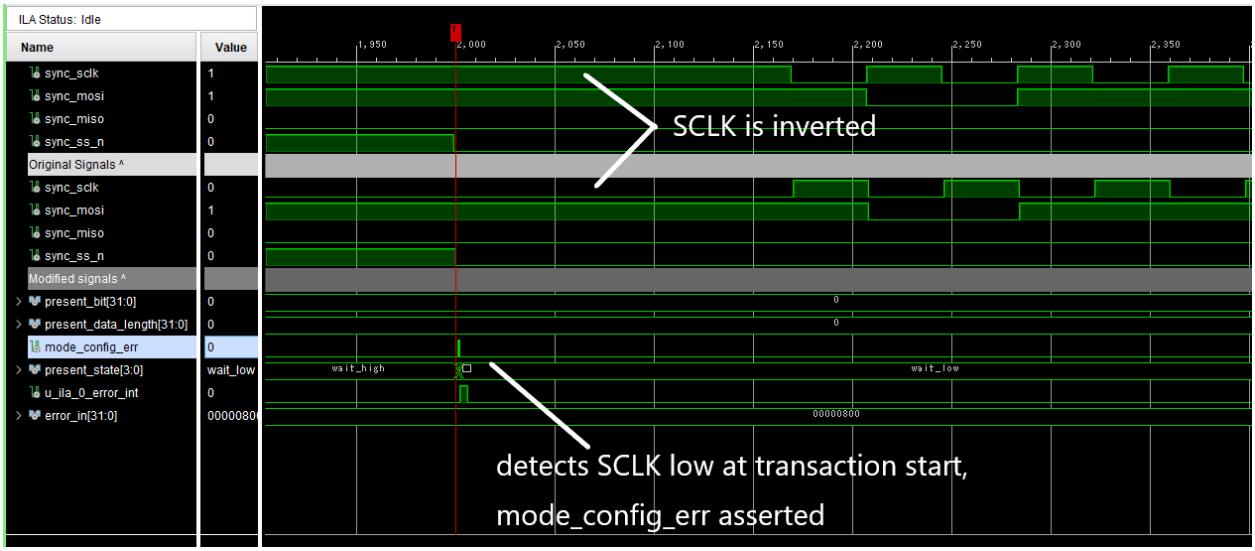


**Figure 5.16** CPOL Mode Fault Injection

The other mode configuration error, dealing with mismatched SCLK polarities, is displayed above in Fig. 5.16. This attack is performed by the SFI by inverting the input SCLK before it is fed to SPI HECAD. The error is detected immediately, as the HRIM ensures the correct clock polarity at the start of a transaction. The SS_N signal drops and the clock polarity by default configurations should be high. It is not, so the HRIM detects this issue and reports it.


**Figure 5.17** SCLK Frequency Fault Injection

The last hardware resource attack performed by the SFI divides the input SCLK down to a different frequency. This attack is displayed in Fig. 5.17. Realistically, no issues should occur from a decreased SCLK frequency if it were the actual signal being fed to the IMU, as the slave will respond to whichever frequency the master sets. As long as it is within a valid range, communication should still occur. The SCLK frequency should be static, however, and any frequency that differs from the expected value is cause for alarm, so the SPI HRIM detects any inconsistencies and reports, as displayed in the waveform above.

## 5.4 I2M Fault Injection

This section will present the results of the SFI performing information level attacks on the SPI I2M submodule. The waveforms will resemble those presented in the previous section in both format and signal content. Additionally signals will be added where necessary to display the attack detection mechanisms. Note that not all possible SFI attacks will be presented, as some detection mechanisms exist as a subset of others. For example, the reasonability checks will not be displayed, as the verification of range checks will exercise both mechanisms. The stuck-at one and zero will be presented, but being stuck at a different value will not because it would trigger for the one and zero conditions as well. The mechanisms will only be presented for one SPI signal, as displaying for all four would only be unnecessarily redundant.

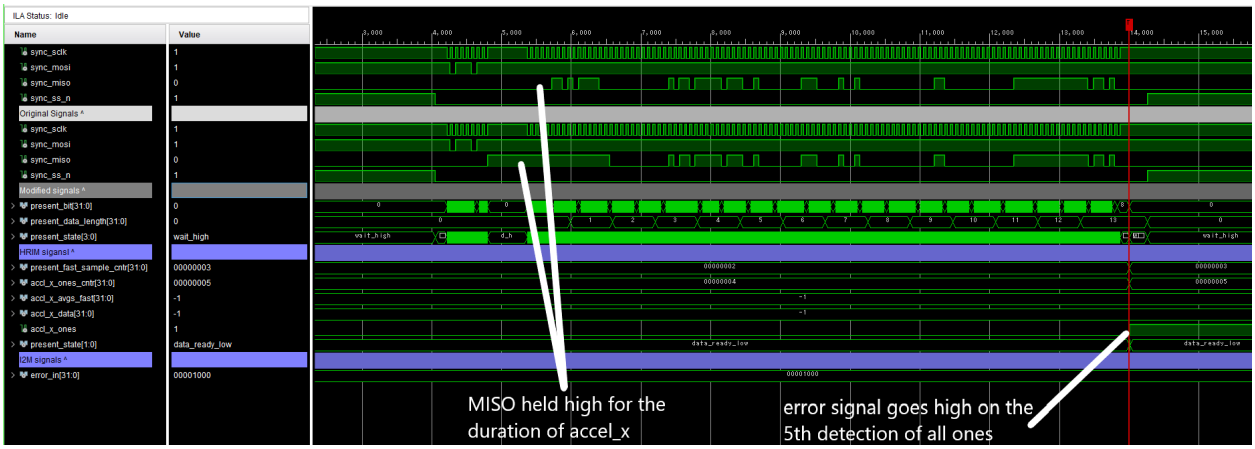


**Figure 5.18** Stuck-At Ones Fault Injection Results

Fig. 5.18 displays the results of the SFI performing a stuck-at ones attack. The attack is performed by keeping the MISO signal high for the duration of the first two bytes of the transaction. These bytes, corresponding to the acceleration reading for the x-axis, will be used for all of the following results. It can also be seen that this attack has been occurring for multiple samples, as the condition is not triggered until a threshold is met. In this case, after the fifth occurrence of all ones, the error signal for the accel_x stuck-at ones is set. In a similar manner, the stuck-at zeros attack is present in Fig. 5.19. Though not visible, the


**Figure 5.19** Stuck-At Zeros Fault Injection Results

generic stuck-at signal for accel_x will also fire due to similar conditions being met.

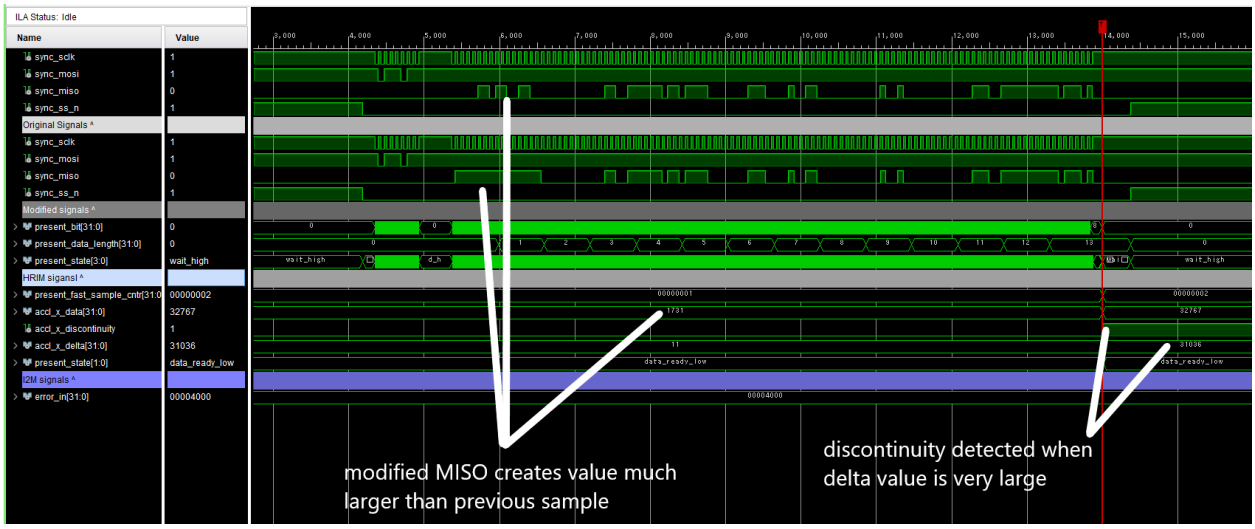


**Figure 5.20** Discontinuity Fault Injection Results

Fig. 5.20 displays the discontinuity fault injection. For this attack, the value of the x-axis acceleration value is drastically changed from one sample to the next. The MISO signal is modified to return a much higher value than the previous sample. After this discontinuity is detected from the large delta value, the corresponding error signal is asserted.

Finally, Fig. 5.21 displays the last waveform in this chapter. The MISO signal is manip-

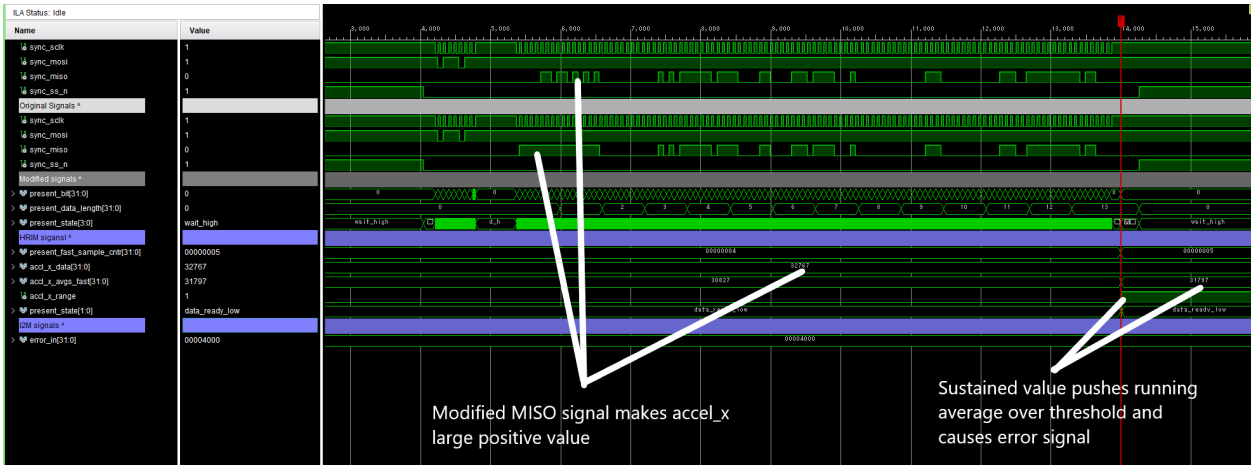


**Figure 5.21** Range Fault Injection Results

ulated to write the maximum possible value for a signed 16-bit number by carefully pulling the value high for all but the first bit in the first byte, which would otherwise be a negative number. This attack is held for several samples, similar to the stuck-at attacks. The running average value is constantly compared against the threshold for a range error. Finally, this condition is met as displayed in the waveform and the error signal goes high. For all of these attacks, though not displayed, the state machine inside the I2M will skip the new data interrupt sent to the FIM, ensuring that any faulty data will not propagate further.

# Chapter Six

# Conclusion and Future Work

UAVs, as they gain popularity in many spaces, have attracted attention as safety-critical systems. The need to secure these systems has become more apparent as more consumer adoption has lead highlighted attack surfaces that pose serious risk to people or the operational environment of the UAV. The HECAD architecture was proposed as a passive monitoring device to catch any atypical operations of a fight controller. Specifically aimed at protecting the VCU Aries FCS, HECAD is developed in an FPGA design to monitor all the bus communication between the flight sensors. At different levels of hierarchy, HECAD achieves this goal by using a priori information about Aries to ensure the sensor hardware, sensor information, and mission are conforming to correct operation.

The IMU on Aries communicates over SPI, a specific bus protocol. HECAD has a specifically implemented component responsible for monitoring this bus and sensor. This component is implemented while satisfying the general requirements of the overall design. For SPI HECAD, several checks are performed at the hardware level to ensure that all communication is correct and no hardware faults have occurred from whatever source. The individual signals are monitored for regular activity, the transaction is well-formed, and that the FCS and IMU are configured correctly. At the information level, a variety of checks are performed to make sure the data does not display any apparent problems. These problems could manifest in large jumps in values, readings running close to the range of the sensor, or values

not changing at all. All of these checks are performed in real time while also passing sensor data up to a central Functional Integrity Monitor.

**Future Work**

The most limiting factor to SPI HECAD efficacy as a monitoring device comes from the threshold values used across the design. These thresholds have been set in accordance to heuristics which serves as a baseline. The functionality of the checks should be fine-tuned once a flight-ready HECAD design is implemented and tested. False-positives should be limited as much as possible by setting the threshold values based on known-good flights.

Additionally, SPI HECAD needs formal integration with the rest of the design. As it was developed in parallel with the other HRIM and I2M modules for other busses and sensors, SPI HECAD could create conflicts with hardware resources inside the FPGA. PL fabric resources are not so much a concern, as discussed in the hardware platform section but other problems may exist. These contain but are not limited to interrupts, AXI interconnect resources, and Zynq PS time slices. These all have to do with how the hardware and information level monitors communicate with the FIM. Until the designs are finalized and the FIM requirements are realized, it is unknown how limited these Zynq resources are and will need to be investigated in future work.

The design itself has not undergone any formal verification or model checking. While extensive ad hoc testing was done, the design should be verified in a model checking utility like Spin or Simulink. This will ensure that SPI HECAD formally performs all that it is required to. Additionally, a model checker will ensure the liveness properties that are critical to several of the state machines inside SPI HECAD. Other analysis would need to be performed on the false positives and negatives once the threshold parameters are tuned. A probabilistic method of comparing the rates of both to correct detections is necessary to ensure the model can correctly capture faulty sensor information.

# REFERENCES


[1] Matthew Leccadito. "A Hierarchical Architectural Framework for Securing Unmanned Aerial Systems". In: *Theses and Dissertations* (Jan. 2017). DOI: https://doi.org/10.25772/0DK3-E418. URL: https://scholarscompass.vcu.edu/etd/5037.

[2] Mark Yampolskiy, Peter Horvath, Xenofon Koutsoukos, et al. "Systematic analysis of cyber-attacks on CPS-evaluating applicability of DFD-based approach". In: Aug. 2012, pp. 55–62. ISBN: 978-1-4673-0161-9. DOI: 10.1109/ISRCS.2012.6309293.

[3] Gaurav Choudhary, Vishal Sharma, Ilsun You, et al. "Intrusion Detection Systems for Networked Unmanned Aerial Vehicles: A Survey". In: *2018 14th International Wireless Communications Mobile Computing Conference (IWCMC)*. ISSN: 2376-6506. June 2018, pp. 560–565. DOI: 10.1109/IWCMC.2018.8450305.

[4] Manuel J. Fernandez, Pedro J. Sanchez-Cuevas, Guillermo Heredia, et al. "Securing UAV communications using ROS with custom ECIES-based method". In: *2019 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED UAS)*. Nov. 2019, pp. 237–246. DOI: 10.1109/REDUAS47371.2019.8999685.

[5] Nils Miro Rodday, Ricardo de O. Schmidt, and Aiko Pras. "Exploring security vulnerabilities of unmanned aerial vehicles". In: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. ISSN: 2374-9709. Apr. 2016, pp. 993–994. DOI: 10.1109/NOMS.2016.7502939.

[6] Andrew Shull. *Analysis of cyberattacks on unmanned aerial systems*. en. Library Catalog: www.semanticscholar.org. 2013. (Visited on 05/19/2020).

[7] Alan Kim, Brandon Wampler, James Goppert, et al. "Cyber Attack Vulnerabilities Analysis for Unmanned Aerial Vehicles". en. In: *Infotech@Aerospace 2012*. Garden Grove, California: American Institute of Aeronautics and Astronautics, June 2012. ISBN: 978-1-60086-939-6. DOI: 10.2514/6.2012-2438. (Visited on 05/11/2020).

[8] Martin Strohmeier, Vincent Lenders, and Ivan Martinovic. "Lightweight Location Verification in Air Traffic Surveillance Networks". In: *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*. CPSS '15. Singapore, Republic of Singapore: Association for Computing Machinery, Apr. 2015, pp. 49–60. ISBN: 978-1-4503-3448-8. DOI: 10.1145/2732198.2732202. URL: https://doi.org/10.1145/2732198.2732202 (visited on 05/11/2020).

[9] Li Teng, Ma Jianfeng, Feng Pengbin, et al. "Lightweight Security Authentication Mechanism Towards UAV Networks". In: *2019 International Conference on Networking and Network Applications (NaNA)*. Oct. 2019, pp. 379–384. DOI: 10.1109/NaNA.2019.00072.

[10] Mukhtar Ahmad, Muhammad Atif Farid, Sheeraz Ahmed, et al. "Impact and Detection of GPS Spoofing and Countermeasures against Spoofing". In: *2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*. Jan. 2019, pp. 1–8. DOI: 10.1109/ICOMET.2019.8673518.

[11] G. Panice, S. Luongo, G. Gigante, et al. "A SVM-based detection approach for GPS spoofing attacks to UAV". In: *2017 23rd International Conference on Automation and Computing (ICAC)*. Sept. 2017, pp. 1–11. DOI: 10.23919/IConAC.2017.8081999.

[12] Mohsen Riahi Manesh, Jonathan Kenney, Wen Chen Hu, et al. "Detection of GPS Spoofing Attacks on Unmanned Aerial Systems". In: *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*. Jan. 2019, pp. 1–6. DOI: 10.1109/CCNC.2019.8651804.

[13] Esat Elezi, Göksel Çankaya, Ali Boyacı, et al. "The effect of Electronic Jammers on GPS Signals". In: *2019 16th International Multi-Conference on Systems, Signals Devices (SSD)*. ISSN: 2474-0446. Mar. 2019, pp. 652–656. DOI: 10.1109/SSD.2019.8893239.

[14] Chang Li and Xudong Wang. "Jamming research of the UAV GPS/INS integrated navigation system based on trajectory cheating". In: *2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*. Oct. 2016, pp. 1113–1117. DOI: 10.1109/CISP-BMEI.2016.7852880.

[15] Anupam Purwar, Divya Joshi, and Vinod Kumar Chaubey. "GPS signal jamming and anti-jamming strategy — A theoretical analysis". In: *2016 IEEE Annual India Conference (INDICON)*. ISSN: 2325-9418. Dec. 2016, pp. 1–6. DOI: 10.1109/INDICON.2016.7838933.

[16] Mahmoud Elnaggar and Nicola Bezzo. "An IRL Approach for Cyber-Physical Attack Intention Prediction and Recovery". In: *2018 Annual American Control Conference (ACC)*. ISSN: 2378-5861. June 2018, pp. 222–227. DOI: 10.23919/ACC.2018.8430922.

[17] An Guo, Dong Yu, Haichao Du, et al. "Cyber-physical failure detection system: Survey and implementation". In: *2016 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*. June 2016, pp. 428–432. DOI: 10.1109/CYBER.2016.7574863.

[18] Victor M. Lopez Rodriguez, Albert Mo Kim Cheng, and Binh Doan. "Work-in-Progress: Combining Two Security Methods to Detect Versatile Integrity Attacks in Cyber-Physical Systems". In: *2019 IEEE Real-Time Systems Symposium (RTSS)*. ISSN: 2576-3172. Dec. 2019, pp. 596–599. DOI: 10.1109/RTSS46320.2019.00073.

[19] Imran Makhdoom, Mehran Abolhasan, Justin Lipman, et al. "Anatomy of Threats to the Internet of Things". In: *IEEE Communications Surveys Tutorials* 21.2 (2019). Conference Name: IEEE Communications Surveys Tutorials, pp. 1636–1675. ISSN: 1553-877X. DOI: 10.1109/COMST.2018.2874978.

[20] Ronald Fernandes, Perakath Benjamin, Biyan Li, et al. "Use of Topological Vulnerability Analysis for Cyberphysical Systems". In: *NAECON 2018 - IEEE National Aerospace and Electronics Conference*. ISSN: 2379-2027. July 2018, pp. 78–81. DOI: 10.1109/NAECON.2018.8556771.

[21] Cheolhyeon Kwon, Weiyi Liu, and Inseok Hwang. "Security analysis for Cyber-Physical Systems against stealthy deception attacks". In: *2013 American Control Conference*. ISSN: 2378-5861. June 2013, pp. 3344–3349. DOI: 10.1109/ACC.2013.6580348.

[22] Garrett Ward. "Design of a Small Form-Factor Flight Control System". In: *Theses and Dissertations* (Apr. 2014). DOI: https://doi.org/10.25772/Z11J-1Z28. URL: https://scholarscompass.vcu.edu/etd/3448.

[23] Joel Elmore. "Design of an All-In-One Embedded Flight Control System". In: *Theses and Dissertations* (Jan. 2015). DOI: https://doi.org/10.25772/63KT-S314. URL: https://scholarscompass.vcu.edu/etd/3981.