NASA/TM–2010-216834

Fault Injection and Monitoring Capability for a Fault-Tolerant Distributed Computation System

Wilfredo Torres-Pomales, Amy M. Yates, and Mahyar R. Malekpour
Langley Research Center, Hampton, Virginia

August 2010

NASA/TM–2010-216834

Fault Injection and Monitoring Capability for a Fault-Tolerant Distributed Computation System

Wilfredo Torres-Pomales, Amy M. Yates, and Mahyar R. Malekpour
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

August 2010

Acknowledgments

The system described in this report originated from an initial experiment concept developed in collaboration with Dr. Oscar R. González and Dr. W. Steven Gray of Old Dominion University. We are grateful for their productive discussions in the development of system requirements.

We would like to thank Sandra V. Koppen and Truong X. Nguyen for their contributions to the definition of the system configuration for fault-injection experiments in a HIRF environment.

The work described here was made possible by the support from Eric G. Cooper, Associate Principal Investigator for NASA's IVHM Project.

The work performed by Old Dominion University as part of this research collaboration is funded by the NASA Langley Research Center under grant NNX07AD52A.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

The Configurable Fault-Injection and Monitoring System (CFIMS) is intended for the experimental characterization of effects caused by a variety of adverse conditions on a distributed computation system running flight control applications. A product of research collaboration between NASA Langley Research Center and Old Dominion University, the CFIMS is the main research tool for generating actual fault response data with which to develop and validate analytical performance models and design methodologies for the mitigation of fault effects in distributed flight control systems. Rather than a fixed design solution, the CFIMS is a flexible system that enables the systematic exploration of the problem space and can be adapted to meet the evolving needs of the research. The CFIMS has the capabilities of system-under-test (SUT) functional stimulus generation, fault injection and state monitoring, all of which are supported by a configuration capability for setting up the system as desired for a particular experiment. This report summarizes the work accomplished so far in the development of the CFIMS concept and documents the first design realization.

Table of Contents

List of Figures

List of Tables

1. Introduction

The system described in this report, henceforth referred to as the Configurable Fault-Injection and Monitoring System (CFIMS), is intended for the experimental characterization of effects caused by a variety of adverse conditions on a distributed computation system running flight control applications. The CFIMS is a product of collaborative research between NASA Langley Research Center (LaRC) and Old Dominion University (ODU) in support of NASA's goal of developing technologies to improve the intrinsic safety of aircraft [NASA06, Shin08]. The research is being performed under the organizational framework of the Aviation Safety Program's Integrated Vehicle Health Management (IVHM) research thrust, which aims to develop tools, technologies and techniques to mitigate hazardous events during flight [IVHM08]. The NASA-ODU collaboration is focused on developing theoretical tools to analyze the relationship between the design features of a representative computational platform and the performance of a flight control system implemented using the platform and operating in harsh environments [Gray08, Gray10, Chávez10].

The distributed computation platform selected for this research is the Scalable Processor Independent Design for Extended Reliability (SPIDER) developed at Langley Research Center under a previous research effort [Miner02, Torres05A]. SPIDER is a concept for a family of general-purpose fault-tolerant architectures that provides a flexible set of design solutions capable of satisfying a wide range of performance and reliability requirements, while preserving a consistent interface to applications. SPIDER has a combination of attributes not found in existing architectures:

- Product family solution adaptable to many different applications;
- Suitable for safety-critical aircraft functions;
- Supports the Integrated Modular Avionics (IMA) architectural concept (i.e., the ability to host many functions of mixed criticality on the same computational platform) [ARINC651];
- Redundancy management decoupled from applications (i.e., the fault tolerance and redundancy management functions required by the applications are handled by the computational platform itself);
- Customizable redundancy management strategy according to the requirements of the applications (i.e., different fault tolerance and redundancy management strategies can be concurrently supported for different applications);
- Byzantine-fault resilience (i.e., the ability to tolerate arbitrary fault manifestations) [Driscoll03];
- Function migration capability (i.e., the ability to dynamically move functions across a network to different computing resources);
- Ability to survive or quickly recover from massive correlated transient upsets;
- Implemented using mostly off-the-shelf hardware and software;
- Support for dissimilar processors;

- Handles part obsolescence (i.e., the design can evolve to accommodate obsolescence of processors and low-level communication hardware);

- High reliability-to-cost ratio; and

- Design assurance based on Formal Methods (i.e., algorithms and design are formally proven correct) [Butler02]

Figure 1.1 illustrates the SPIDER topology and main system services. The architecture consists of processing elements (PEs) executing the applications and other high-level system functions, and the ROBUS (Robust Bus) communication system, which provides basic services as the foundation for higher-level functions. The data network has a redundant active-star topology with the Bus Interface Units (BIUs) serving as the access ports and the Redundancy Management Units (RMUs) providing connectivity as network hubs. All the communication interfaces are bi-directional. The links between BIUs and RMUs form a complete bipartite graph in which each node is directly connected to every node of the opposite kind. A PE can be collocated and tightly coupled at the physical level to its corresponding BIU (i.e., they can share a fault-containment region (FCR)) [Lala91], or they can be physically independent components.

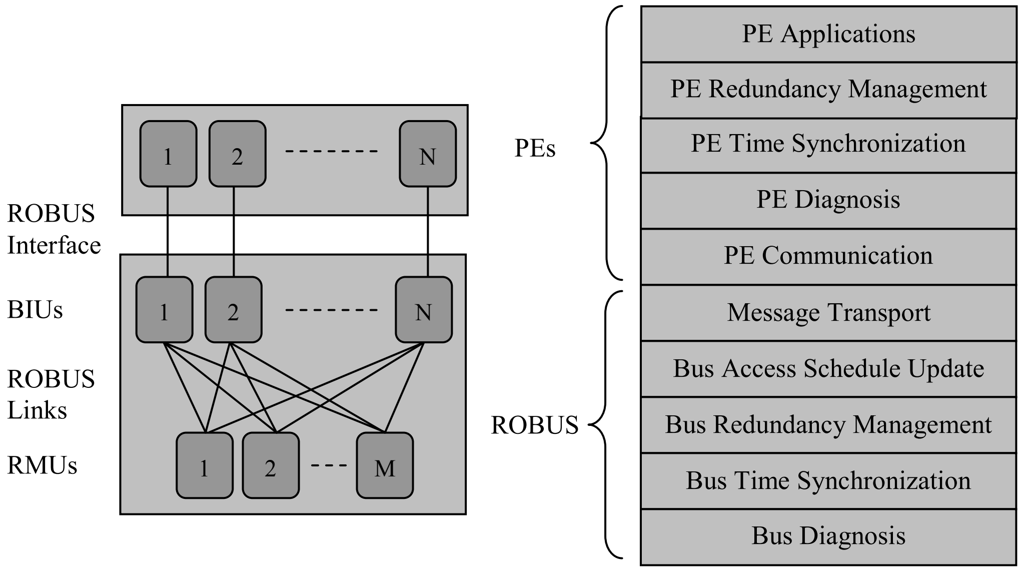


Figure 1.1: SPIDER topology and services

ROBUS implements basic distributed agreement protocols at the level they are most effective (i.e., in hardware) and is intended to reduce the computational burden on the PEs by providing a simple communication system abstraction for what is an inherently complex distributed processing problem. ROBUS-2, an instance of ROBUS, is a time-division multiple access (TDMA) broadcast data communication system (i.e., a data bus) with media access control by means of a time-indexed communication schedule. ROBUS-2 provides guaranteed fault-tolerant services to the attached PEs in the presence of a bounded number of internal faults. These services include message broadcast (Byzantine Agreement), dynamic communication schedule update, time reference (clock synchronization), and distributed diagnosis (group membership). ROBUS-2 also features fault-tolerant startup and restart capabilities. ROBUS-2 tolerates internal as well as PE faults and incorporates a dynamic self-reconfiguration capability driven by the internal diagnostic system. ROBUS-2 consists of custom-designed hardware-based ROBUS Protocol Processors (RPPs) implementing the ROBUS-2 functionality, and a lower-level physical communication network consisting of full-duplex data links [Stallings94] interconnecting the RPPs (see ROBUS Links in Figure 1.1). Additional information about ROBUS-2 can be found in [Torres05A] and [Torres05B]. For completeness, Section 3 of this report provides an overview of ROBUS-2. The VHDL [Armstrong93] source code for the ROBUS-2 RPP is publicly available on the Internet with an open-source license agreement [R2PP].

In the NASA-ODU research collaboration, the range of adverse events to be considered are those characteristic of environmental threats like high-energy particle radiation [Taber93, Zhang08, Zhang09] and electromagnetic interference (EMI) from sources such as lightning and high-intensity radiated fields (HIRF) [Fuller95, Gray00, González01]. These environments have the potential to cause random fault manifestations in individual avionics components and to generate simultaneous system-wide faults that can overwhelm existing resource management mechanisms [Gray08, Hess97]. A flight control system using SPIDER will be subjected to physical and simulated faults in controlled laboratory conditions [Arlat89, Arlat03, Hsueh97, Torres08A] while gathering data suitable for characterizing fault effects at the control system level and at the computation platform level. The CFIMS is a critical element of the research effort as the means to generate experimental data for the development and validation of performance models for operation in adverse conditions.

The purpose of this report is to summarize the work accomplished so far in the development of the CFIMS concept and to document the design of its first realization. The next section describes the general concept for the CFIMS capability. That is followed by an overview of the ROBUS-2 design. After a high-level description for the current CFIMS, the report provides a description of the CFIMS hardware and software elements. A summary and remarks about future work conclude the report.

2. Concept for a Configurable Fault-Injection and Monitoring Capability

Relative to the primary goals of the NASA-ODU collaboration, the CFIMS is the main research tool for generating actual fault response data with which to develop and validate analytical performance models and design methodologies for the mitigation of fault effects. It is expected that achieving the project goals will involve a combination of systematic exploration of the problem space and an iterative process of model refinement involving experimentation and data analysis. Given the uncertainty about how this research will evolve, putting together a-priori a comprehensive list of well-defined design requirements for the CFIMS is a difficult (if not impossible) task with a high likelihood for an unfavorable outcome. Thus, rather than a fixed design solution, we envision a flexible system that can be adapted to meet the evolving needs of the research. The CFIMS should be an enabling tool that allows the investigators to focus on the research problem without being hampered by system limitations due to prior design choices.

With this aim in mind, the first step in the development was to define a concept that captured the essence of what is needed. Figure 2.1 shows a graphical depiction of the concept. The central element is the system under test (SUT), which is a configurable version of SPIDER with added functionality to support simulated fault injection and state monitoring. The CFIMS controls the environment within which the SUT operates. The capabilities of the CFIMS include functional stimulation of the SUT, fault injection, and state monitoring, all of which are supported by a configuration capability for setting up the system as desired for a particular experiment. An experiment is defined by the specifications of the system configuration, the SUT workload, and the faultload to which the SUT will be subjected. Basically, the purpose of an experiment is to gather observations about the response of the SUT when operating under conditions determined by the specifications of the configuration, workload and faultload.

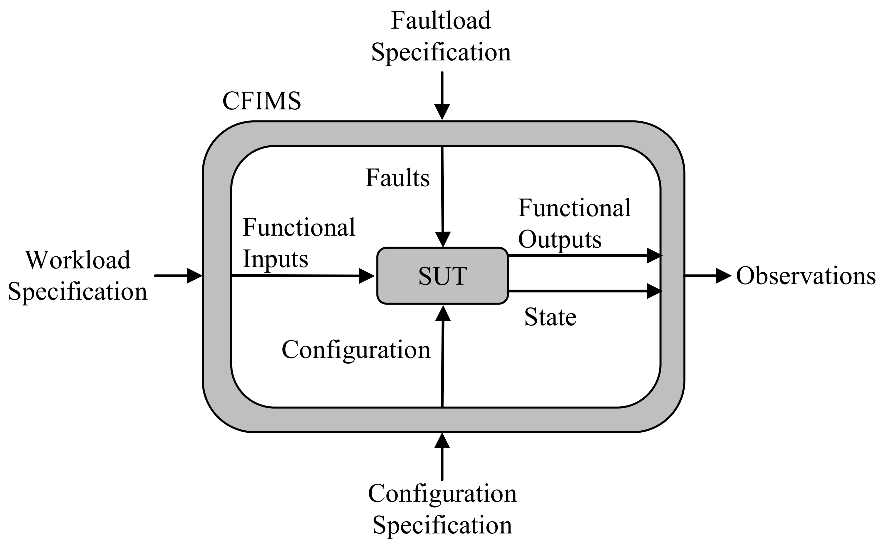


Figure 2.1: Main components and information flows in the configurable fault injection and monitoring capability

In general, the SUT is a distributed computation system executing an application. Figure 2.2 illustrates the role of the SUT in a flight control system and shows a generic SUT functional architecture at the application level consisting of a set of intercommunicating processes. The definition of the application processes and the inter-process communication, as well as their mapping to the resources of the computation platform, are determined by the configuration specification. In SPIDER, all the application processes run on the PEs, which also handle the communication among the processes by leveraging the lower level services provided by ROBUS.

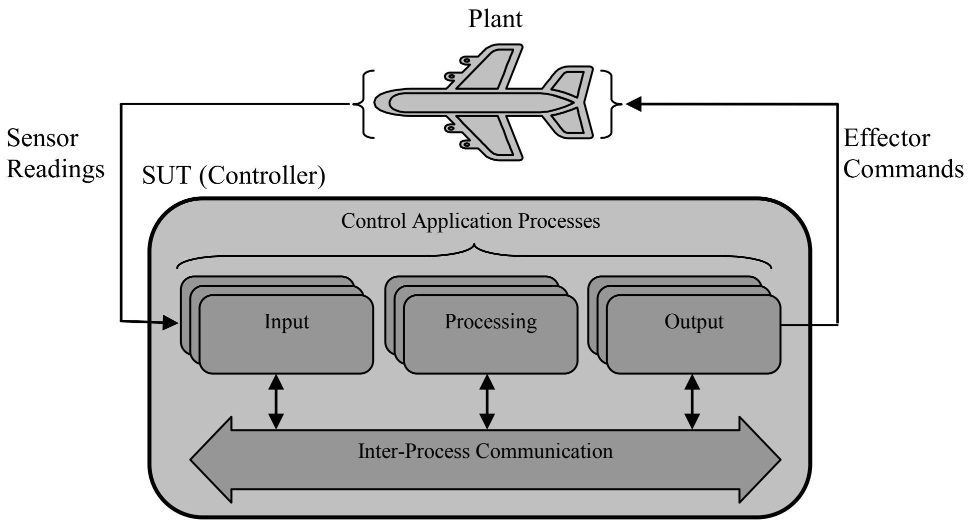


Figure 2.2: Application-level view of the SUT configured as the flight controller

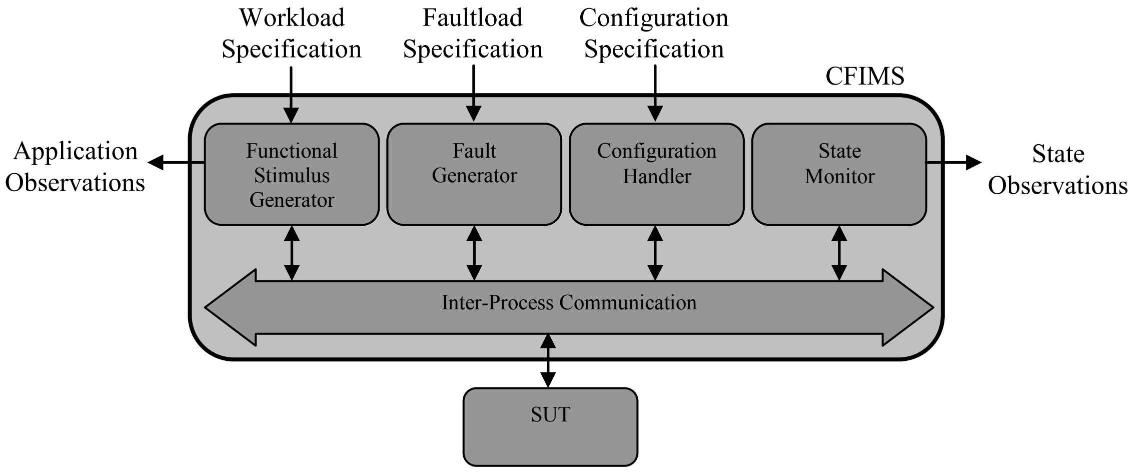


Figure 2.3: High-level view of the CFIMS functions

At a high level, the CFIMS can be viewed as a distributed set of intercommunicating processes (or modules) as illustrated in Figure 2.3 where the SUT is depicted as an external communicating process. The modularization of the CFIMS affords a reduction in the complexity of the development by containing most design changes within individual functional modules and limiting the interaction between modules to clearly defined interfaces. The function of the Configuration Handler process is to setup the system, including the CFIMS and the SUT, as indicated in the configuration specification. This configuration capability is one of the features that allow us to adapt the system to changing requirements. The Functional Stimulus Generator process is responsible for implementing the plant functionality to be controlled by the application processes running on the SUT as shown in Figure 2.2. The Functional Stimulus Generator also has the capability to monitor the behavior of the application and generate observations for post-test analysis. The Fault Generator process injects faults into the SUT according to the faultload specification. The injected faults can be actual physical faults at the hardware level or simulated faults with similar manifestations. The State Monitor process collects state data from the SUT and outputs another stream of observations for post-test analysis. The following subsections elaborate on the desired features for the SUT and the CFIMS.

2.1. SUT Functional Stimulation and Monitoring

This aspect of the system determines the application implemented by the SUT and the CFIMS. As previously described, the type of application of interest is a closed-loop flight control system with the SUT performing the role of the controller and the CFIMS implementing the controlled object (or plant). However, given that the SUT is a general purpose computer and that there may be circumstances in which we may want to have a very simple application layer and focus on the fault effects on the lower level services of the computation platform, it is advantageous to conceptualize the function performed by the SUT as a generic application consuming inputs and generating outputs. The CFIMS provides the workload for the SUT, receives the generated functional outputs, and gathers observations about the response to injected faults.

In general, the mechanisms for testing (or "exercising") the function of the SUT should be scalable and flexible to accommodate a wide range of sizes of the computation platform, applications with various degrees of complexity, and fine precision for the specification of functionality for the system response monitors. There should also be mechanisms to allow various degrees of precision in the coordination between the function testing activities, the injection of faults, and the collection of application and state observations. The CFIMS should also provide the means to easily correlate the data in the output observation streams.

2.2. Fault Injection

In the NASA-ODU collaborative research, only physical hardware faults (i.e., faults involving the physical parts of the system) are of interest. For the purpose of this research, it is assumed that software and design faults are not significant factors in the behavior of the SUT. In essence, a (physical hardware) fault is a physical event that causes a malfunction (i.e., a deviation from correct operation) in a system component. A system is composed of a collection of components organized such that their interaction generates a behavior implementing the function described in the system specification [Avizienis04]. The components of a system are themselves systems that implement lower-level functions.

The CFIMS fault injection capability is intended to force anomalous behavior on the components of the SUT. It is expected that for every SUT function and state monitoring setup, there will be many injection rounds with a variety of fault patterns, some of which may target multiple nodes, sometimes simultaneously. The fault injection configuration is likely to be the configuration that will change most often and the one to require the most flexible and precise solution for specification and execution of a test. Following the terminology in [Arlat03], some of the desired properties for a fault injection capability to assess the effectiveness of a fault-tolerant system include the following:

- Reachability: the degree to which the possible fault locations can actually be reached by the means of injection;

- Controllability: the ability to control the location and time at which faults are injected (i.e., the precision with which faults can be injected);

- Repeatability: the ability to repeat fault injection experiments with high accuracy;

- Reproducibility: the ability to reproduce results in a statistical sense (which does not imply highly accurate fault repeatability); and

- Non-intrusiveness: the ability to avoid or minimize the impact of the fault injection instrumentation on the behavior of the SUT.

The injection of faults will normally be seen as the main independently controlled variable in an experiment. Figure 2.4 illustrates in concept the elements of the CFIMS fault injection capability for the generation of a faultload based on the injection specification and how the manifestations of the injected faults propagate and can be observed (and analyzed) at various SUT abstraction levels. In the CFIMS, application-level observations will capture fault manifestations at the external SUT interfaces, and the state monitoring capability will gather observations about fault manifestations within the SUT. As shown in Figure 2.4, the fault injection capability supports injection of physical and simulated faults. Physical fault injection will be effected indirectly by controlling the characteristics of environmental disturbances, including high-energy particle radiation and high intensity electromagnetic fields, generated in controlled laboratory conditions [Clough96, FAA93, Taber93, Torres08A]. In general, fault injection based on particle radiation has high reachability, but controllability and repeatability are low [Arlat03, Karlsson95]. Electromagnetic radiation also has very low controllability and repeatability, and in addition, reachability is lower than with particle radiation. Thus, in general, these environments are not the most effective for achieving the desirable attributes for a fault injection capability intended for a thorough characterization of effects. Nevertheless, these environments are representative of the types of conditions experienced by operational systems and that makes the experimental results immediately applicable to the goals of the NASA IVHM project. The data from experiments in these environments can be used to characterize the bounds of the relevant fault space, which can then be methodically explored in detail with simulated fault injections.

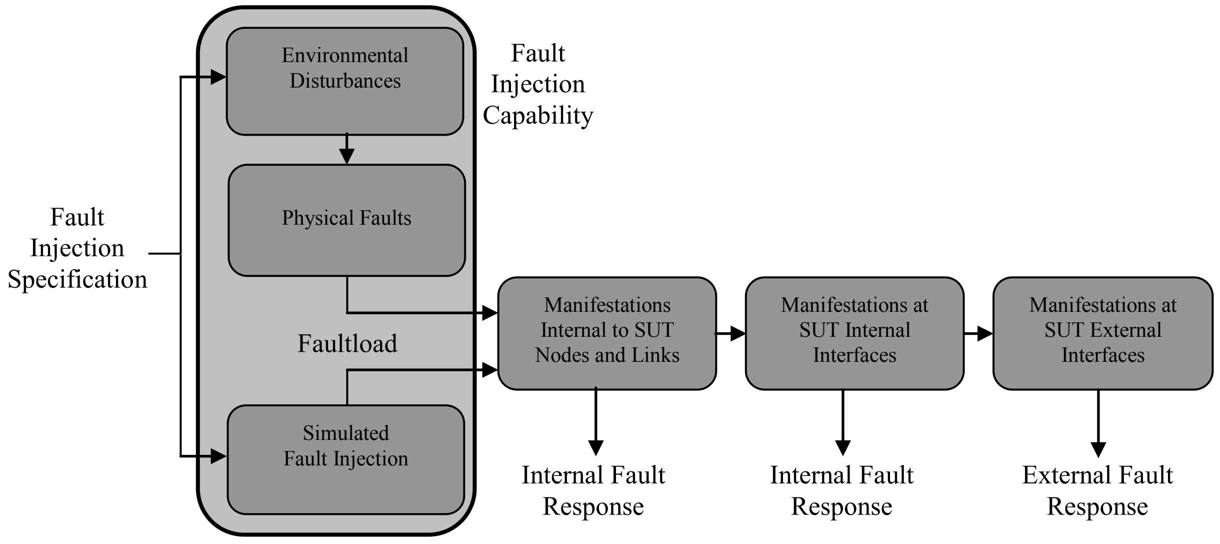

Environmental Disturbances
Fault Injection Capability
Physical Faults
Fault Injection Specification
Faultload
Manifestations Internal to SUT Nodes and Links
Manifestations at SUT Internal Interfaces
Manifestations at SUT External Interfaces
Simulated Fault Injection
Internal Fault Response
Internal Fault Response
External Fault Response

Figure 2.4: Faultload generation and propagation of fault manifestations

In general, the system fault space can be large and complex when measured in terms of the possible fault locations with respect to the actual smallest physical parts that make up the system. A strong determining factor for the complexity of a fault effects analysis is the selection of the components that constitute the smallest units of failure relevant to the analysis. In the SPIDER-based distributed SUT, the nodes and their communication links are the smallest recognized entities in the distributed interaction protocols, including resource management and agreement protocols. The system nodes and links are natural boundaries that can be leveraged in the definition of the minimum component granularity for fault effects analyses.

At their highest structural level, every SUT node is composed of a communication component and a computation component (see Section 3 in this report and [Torres05A]). The communication component of a node manages all the messaging interfaces to other nodes. The computation component performs all the local data processing according to the definition of the distributed protocols and the system application. A goal in the design of the SPIDER fault-tolerant system is to contain the internal propagation of faults and errors to allow the delivery of correct services at the external interfaces. As part of a node's error handling capability, the communication and computation components perform acceptance checks (e.g., CRC checks [Stallings94] or time of arrival checks [Torres05A]) on received messages with the intent of impeding the propagation of errors from other FCRs and minimize the likelihood that such errors corrupt the state of the receiving node. It is possible to significantly reduce the fault space to be analyzed by focusing on the manifestations of a transmitting node's behavior as perceived by a receiving node taking into consideration the results of error checks at the receiving node. The space of all possible faults that can occur at a transmitting node and its communication link can be mapped to a simple classification by leveraging the concept of error detectability (which is related to the property of integrity, i.e., the absence of improper state alteration) [Avizienis04, Paulitsch05] to establish equivalence relations on the manifestations at a receiver. The fault classes in this model for communication between a sending node and a receiving node are labeled Good, Omissive and Transmissive (GOT). Alternatively, the classes can be respectively labeled Correct, Detected and Undetected (CDU) (see [Torres08B]). In this GOT/CDU model, the fault categories are defined as shown in Table 2.1, where correctness is determined by the system specification. This fault model for one-to-one communication can be used through composition to study the propagation of errors and error detection in a large distributed system network.

Table 2.1: Detectability-based fault model for one-to-one communication

Fault Mode	Description
Good/Correct	The receiver accepts a correct message (i.e., there is no fault or fault manifestation).
Omissive/Detected	The receiver detects a missing message or rejects an incorrect message.
Transmissive/Undetected	The receiver accepts an incorrect message.

A fault model better suited to the analysis of properties in distributed agreement protocols is the omissive-transmissive hybrid (OTH) model [Azadmanesh00, Torres08B, Weber06], which adds the concept of consistency of perception to the definition of the fault categories. In this model, the smallest system component is a sending node and its transmission links to all the nodes receiving messages from it (i.e., the observers). The receivers either agree on their observations (i.e., their observations are symmetric) or they do not (i.e., their observations are asymmetric). The fault categories in this model are defined in Table 2.2. The OTH model allows a further reduction in complexity by minimizing the fault space to a small set of categories that are relevant to the analysis of distributed agreement properties.

Table 2.2: Omissive-Transmissive Hybrid fault model for one-to-many communication

Fault Mode	Description
Correct	All observers receives the same correct message.
Omissive Symmetric	Each observer declares the message invalid, either because the message was not received or it was detectably incorrect.
Transmissive Symmetric	Each observer accepts the same incorrect message.
Strictly Omissive Asymmetric	Some observers receive the same correct message and others declare the message invalid, either because they do not receive a message or declare invalid their received message.
Single-Data Omissive Asymmetric	Some observers accept the same incorrect message and others declare the message invalid, either because they do not receive a message or declare invalid their received message.
Transmissive Asymmetric	The observers have other patterns of disagreeing observations.

An enhancement to the OTH model is to add the dimension of time duration to the fault categories. With the concept of persistence of a fault, each OTH category can be subdivided by fault duration categories defined according to the particular conditions of the problem at hand (e.g., transient and permanent duration). We refer to this classification as the Detectability, Consistency and Persistence (DCP) fault model.

A peculiarity of these observer-based fault models is that the classification of any given physical fault is not absolute but relative to the states of the observers, which normally change during system operation. This may limit the achievable reduction in the fault space complexity.

These fault models can be used to characterize the faults in the physical fault injection experiments. In addition to determining the proportion of faults within each category of the selected fault model, the characterization must also include the modeling of the fault activation pattern in the time domain. The characterization of physical fault effects can then be used as a reference to develop a library of simulated faults that is in some sense equivalent to the actual physical faults. The simulated faults in the CFIMS are injected by introducing errors in the behavior of SUT components. Multiple communication and computation components within a node can be targeted for injection depending on the desired fault manifestations.

An issue that will need to be addressed in the research is how to model the effects of transmissive (i.e., undetected) faults, whose actual impact on the behavior of a node and the system is likely to be dependent on the particular state of a receiving node at the time of arrival of propagated errors. ROBUS uses voting in its distributed protocols to mask the effects of transmissive faults. The degree of fault tolerance in ROBUS (i.e., the number of faults it can tolerate) is determined by the available redundancy and number of active transmissive faults in the system [Torres05A]. The ROBUS distributed diagnosis service allows the system to continue coordinated operation by the exclusion of diagnosed faulty nodes from participation in distributed decisions. In addition, ROBUS provides a bus failure detection and re-initialization capability to restore coordinated operation after events of massive correlated transient faults that exceed the fault tolerance capabilities of the network [Torres05A]. However, all these mechanisms depend on the detection of abnormal operational conditions. It is uncertain what the behavior of the system would be if its fault handling mechanisms were overwhelmed by a large number of undetected faults.

2.3. State Monitoring

The state monitoring capability is intended to provide observability into the internal operation of the SUT. This capability should provide the means to collect data of adequate fidelity to perform detailed post-test event analyses and modeling of physical faults and fault manifestations. Given the functional and physical distributed nature of the SUT, the state monitoring capability must gather data from many different locations and package it into coherent observations. There will be observations of internal activity at every node and of interactions between nodes at their interfaces. At a higher level, the state monitoring capability must provide the means to precisely correlate the distributed system observations to allow the analysis of causality in node and system event sequences. At a minimum, the timing of all the observations should be referenced to a common timeline. The added state monitoring instrumentation should not interfere with the execution of the SUT, but it will require support in the design of the SUT to enable direct access to state information at its source. The state monitoring capability should be able to properly handle operation of the SUT under the influence of injected faults as well as normal SUT operation. This capability should be easily configurable to select the degree and amount of detail in the observations and to specify complex observation-triggering conditions.

2.4. System Configuration

Although our knowledge of the research objectives allows us to develop abstract designs for the SUT and the CFIMS, we assume that because of the uncertainty about the future direction of the research, it is highly improbable that a single SUT and CFIMS implementation of reasonable complexity will be adequate to meet the research needs over the duration of the project. The configuration capability is intended to provide a flexible way to specify the functional and physical characteristics of the SUT and the CFIMS to meet evolving research needs. Essentially, what we need is the capability to change the system anywhere from the highest level architectural definition to the lowest implementation details.

The chosen configurability solution consists of custom system designs with hardware and software components coded in-house and running on a generic reconfigurable execution platform for distributed systems. The hardware and software code to be developed will be highly parameterized for structure and behavior, with some parameters specified at synthesis or compilation and others at runtime. The computing nodes of the generic reconfigurable platform have programmable computation and communication resources with which to build specialized networks of interconnected nodes. The computation resources include general purpose processing for software functions and programmable hardware resources based on FPGAs (Field Programmable Gate Arrays) for hardware-implemented functions. The communication resources provide basic mechanisms that enable the nodes to interact by messages (see Section 4).

The specifications of the execution platform establish fundamental constraints on the capabilities of systems implemented on it. So, in addition to allowing the reconfiguration of the system, the code parameterization gives it reusability and portability in case there is a need to change the execution platform.

The attributes of the SPIDER architecture and the ROBUS-2 communication system [Torres05B] (e.g., physical modularity, layered services and time-triggered operation) makes them especially well suited to support this system configuration concept.

This configurability approach minimizes the limitations on the systems that can be specified by allowing modifications to be made at any level, including design, synthesis, and runtime. There is, however, an associated fundamental tradeoff in that, although higher level changes allow more flexibility in the configuration of the system, there is a cost increase in the time and effort needed to implement those changes. This offers a motivation for maximizing the synthesis and runtime configurability of the system to avoid the complications of making changes to the design itself. However, in the process of design development and implementation, careful consideration should be given to how a design is likely to be used and the available time for development so as to properly select the optimum balance between configurability and complexity of the design.

3. Overview of the ROBUS-2 Communication System

This section presents a brief description of ROBUS-2, including the structure and operation of the system. The existing ROBUS Protocol Processors [R2PP] are intended for systems with a relatively small number of PEs due to the significant size-complexity of the RPP's implementation. More detailed information about ROBUS can be found in [Torres05A] and [Torres05B].

3.1. System Structure

Figure 3.1 shows the ROBUS topology. The bus has an active-star architecture with the Bus Interface Units (BIUs) serving as the bus access ports and the Redundancy Management Units (RMUs) providing connectivity as network hubs. The network between BIUs and RMUs forms a complete bipartite graph in which each node is directly connected to every node of the opposite kind. Only the links shown are available for communication. All the communication links are bidirectional.

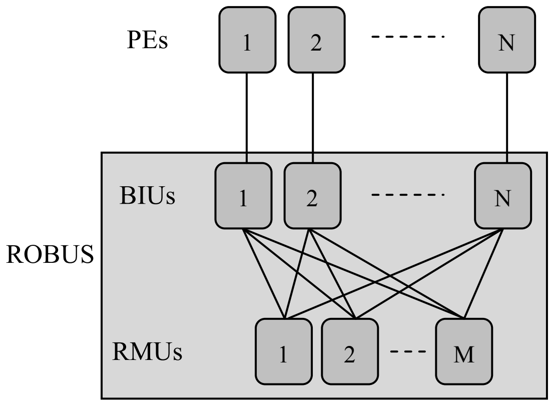


Figure 3.1: ROBUS topology

Figure 3.2 depicts the basic structural components of BIU and RMU nodes. The Communication Module handles all the point-to-point communication. The links between BIUs and RMUs can be either one-to-one or one-to-many links, as long as broadcast communication is supported. The nature of the links between BIUs and PEs depends on how they are physically related. If a BIU and its corresponding PE are physically independent, then they are interconnected by a one-to-one data communication link. If a PE-BIU pair are physically integrated (e.g., same printed circuit board), then some other means of local data exchange can be used (e.g., a dual port memory).

The Computation Module, also known as the ROBUS Protocol Processor (RPP), handles all the ROBUS-specific functions including mode transition logic, low-level protocols, error detection, diagnosis, reconfiguration, and distributed coordination. The main difference between BIUs and RMUs is the functionality of their RPPs. Normally, the main determining factor for the number of BIUs in a ROBUS implementation is the number of PEs needed to meet functional and reliability system requirements. The number of RMUs is determined by the system reliability target and is independent of the number of BIUs or PEs.

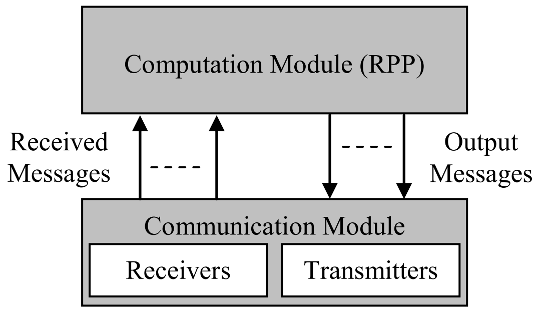


Figure 3.2: Generic top-level node structure for BIUs and RMUs

3.2. Distributed Coordination

Each ROBUS node is driven by an independent, free-running physical oscillator. These oscillators are characterized by known upper and lower bounds on their drift rates with respect to real time. Each node also has a logical-time clock, referred to as the local-time clock, which keeps track of the passage of time as indicated by the physical oscillator. Given an initial precision of synchronization for the local times at any two nodes, the precision can worsen over time at a rate determined by the drift rate bounds of the physical oscillators.

There are two main categories of ROBUS protocols: synchronization and synchronous. The synchronization protocols use event-triggered communication and event-processing operations to generate high-precision distributed events that are used to synchronize the local-time clocks. The synchronous protocols use the synchronized local-time clocks to process information using time-triggered communication and operations. To achieve proper coordinated action in the execution of the synchronous protocols, the local-time clocks of the participating nodes must be synchronized within some known bounded precision.

ROBUS has two synchronization states: synchronized and unsynchronized. In the synchronized state, the precision of synchronization is determined by an internal distributed reference event generated by a clock synchronization protocol. The precision of this event allows the nodes to achieve very tight local-time synchronization. The bus is in the unsynchronized state when it transitions to the startup and restart processes. The precision of synchronization in this state is mainly determined by events not directly controlled by the bus. It is assumed that the synchronization precision in this mode has a known bound that can be large relative to the precision in the synchronized state. The bus transitions from the unsynchronized state to the synchronized state after the execution of a synchronization protocol. Because the local times can drift apart, a synchronization protocol must be re-executed at regular intervals to ensure that the local times are kept synchronized. The rate of re-synchronization is constrained by physical parameters of the design (e.g., oscillator drift rates) as well as precision and accuracy goals. The fault-tolerance attribute of the synchronization protocols enables the bus to achieve and maintain synchronization even in the presence of failed nodes.

The execution of synchronous protocols is driven by the local time and a time-indexed operation schedule. The low-level distributed protocols specify the system activities by defining the active nodes, operations, operation sequencing, and message flow patterns for each operation. The timing of operations is determined using a model of distributed synchronous composition. This execution scheme and the high synchronization precision in the synchronized state make the steady-state behavior of ROBUS highly deterministic as it precisely specifies the timing of all the internal communication between BIUs and

RMUs, as well as the communication with the PEs.

3.3. Redundancy Management

The purpose of redundancy management is to increase the probability of continued service delivery through effective utilization of available resources. ROBUS is designed to manage its redundant BIU and RMU components independently from the PEs.

Fault containment refers to the isolation of physical faults to prevent their propagation throughout the system. This is achieved by establishing fault containment regions (FCR) that ensure a sufficiently high degree of independence with respect to physical faults. Physical system components within an FCR are considered to experience correlated faults because the cause of the faults can affect multiple components simultaneously or because faults originating in one component can propagate to others within the FCR. Thus, if any component within an FCR is affected by a fault, every component and function within the FCR is considered untrustworthy. Ideally, the FCRs have independent power supply and are physically and electrically isolated from each other. Communication between FCRs is through carefully specified interfaces that ensure a sufficiently high degree of fault containment. In ROBUS, each RMU node is contained in its own FCR, and each BIU can be located by itself in a separate FCR or it can share an FCR with its corresponding PE.

Each BIU and RMU node is an observer of every node on the bus. An observed node is referred to as a defendant. The diagnostic system of ROBUS is a distributed system divided into two layers. In the local layer, the nodes monitor the communication and independently diagnose each individual node and the bus as a whole. In the collective layer, the nodes exchange local diagnostic information to augment their local assessments. Every ROBUS node performs the diagnostic functions of error detection, node diagnosis, and bus diagnosis.

Error detection is the foundation of the diagnostic system. The following list covers all the categories of error checks performed by the ROBUS nodes. These checks generate the syndromes from which diagnostic decisions are made.

- Communication checks monitor the communication links between the nodes.
- In-line checks are applied to received messages and are based on expected timing and content characteristics.
- Cross-lane checks also detect errors in received messages by comparing them against the result of dynamic voting.
- Protocol checks inspect received messages and voting results with respect to expected properties for intermediate and final protocol results.
- Self-checks are performed by a node to monitor its own operation.
- PE checks inspect the messages received by a BIU from its corresponding PE.

The diagnostic system assesses each node to determine its suitability to participate in the delivery of services to the PEs. A trustworthy node can be relied upon to deliver the expected services. Untrustworthy nodes do not behave as expected. A defendant is locally accused by an observer when the observer determines that the defendant is untrustworthy but it is uncertain whether other observers have reached the same conclusion. A defendant is collectively convicted when the observers agree that a sufficient number of them consider the defendant untrustworthy. An observer forms a full diagnostic assessment of a defendant based on local and collective diagnoses.

In the context of ROBUS, a clique is a group of BIUs and RMUs working together in a coordinated way to deliver services to the PEs. A clique is considered trustworthy if its services are in accordance with the ROBUS functional specification [Torres05A]. The diagnosis of the bus consists of determining if exactly one trustworthy clique is in operation.

The BIU and RMU nodes use the diagnostic assessments to determine the clique membership. A clique is reconfigured by adding or removing nodes from its membership. The purpose of reconfiguration is to enhance the ability of a clique to establish and preserve proper service delivery in the presence of untrustworthy nodes. A clique member is allowed to participate in the delivery of services to the PEs and is referred to as a trusted node. A node searching for and trying to become part of a clique is called a recovering node.

The FCRs ensure that the only error propagation path between nodes is through their interfaces. Error containment for the interfaces between BIUs and RMUs is realized by placing barriers at both ends of each interface. The BIUs and RMUs disable their outputs upon detection of a local failure or a bus failure (i.e., they implement a fail-stop failure response). At the receiving end of the interfaces, the nodes use input-error detection (e.g., communication and in-line checks) and dynamic voting to block propagated errors from untrustworthy sources. The sources whose inputs are considered in a vote are called the eligible voters.

3.4. Operational Modes

Figure 3.3 shows the major mode transitions for BIU and RMU nodes. A recovering node is in a mode other than Clique Preservation. After a power-on enable, a recovering node goes to the Self-Test major mode to perform a local initialization and test its circuitry. The recovering node will remain in this mode indefinitely unless it successfully passes the test. After completing the self test, the recovering node enters the Clique Detection mode to determine if there is a clique operating in the Clique Preservation mode. If a clique is found, the recovering node transitions to the Clique Join mode, where it demonstrates to the clique members that it is suitable for admission. If a clique is not found, the recovering node transitions to the Clique Initialization mode to form a new clique. Upon successful completion of the recovery process by either joining an existing clique or forming a new clique, a node transitions to the Clique Preservation mode where, as member of a clique, it delivers services to the PEs according to the ROBUS service specification [Torres05A]. At any time, if a node detects a local failure or a bus failure, it transitions back to the Self-Test mode to reinitialize its operation and find nodes suitable for providing communication services to the PEs.

3.4.1. Clique Preservation

Figure 3.4 illustrates the minor mode transitions for the Clique Preservation major mode. In the Schedule Update mode, a schedule-download protocol is executed to allow the PEs to reprogram the bus communication schedule according to their needs. During PE Communication, first the PE messages are broadcast according to the communication schedule, and then the BIUs and RMUs exchange accumulated accusations against nodes of the opposite kind, which serves to enhance the diagnosis and reconfiguration capabilities of the bus. This is followed by a re-synchronization of the local time in the Synchronization

Preservation mode and then a reassessment of the clique membership in the Collective Diagnosis mode.

3.4.2. Self-Test

Upon entering the Self-Test mode, a node disables its output and performs a local hardware reset. This mode serves as a checkpoint in which the nodes are required to exercise and assess the status of their circuitry before attempting to join other nodes on the bus. This mode also provides a safe state to which ROBUS nodes can go after detecting a failure and before attempting to re-engage.

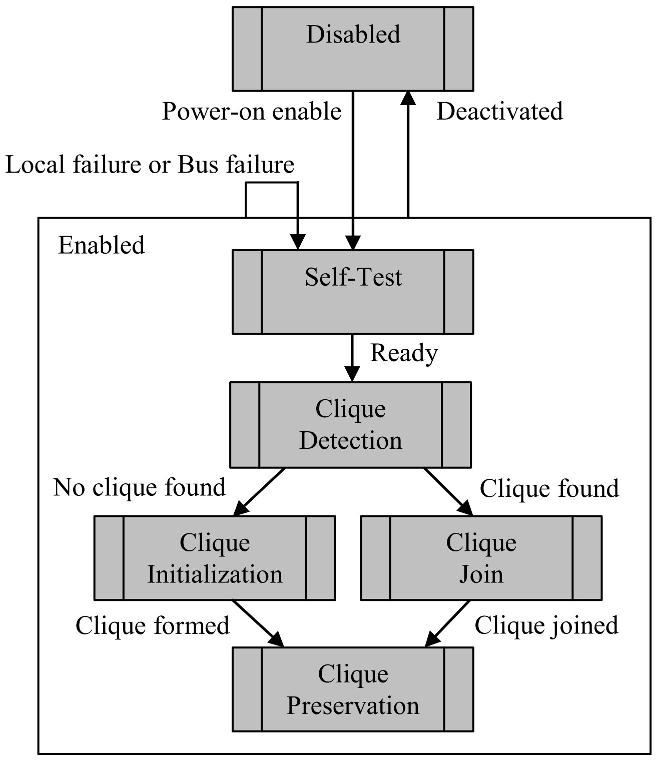


Figure 3.3: Major operational mode transitions for ROBUS nodes

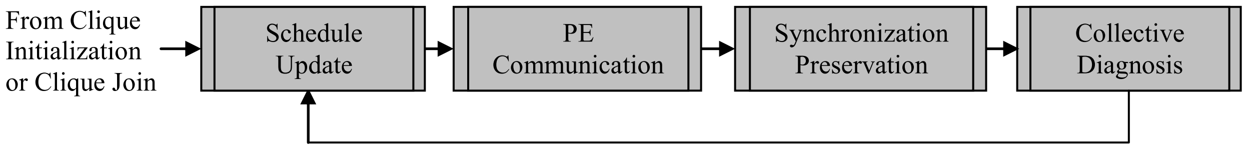

From Clique Initialization or Clique Join
Schedule Update
PE Communication
Synchronization Preservation
Collective Diagnosis

Figure 3.4: Minor mode transitions for Clique Preservation mode

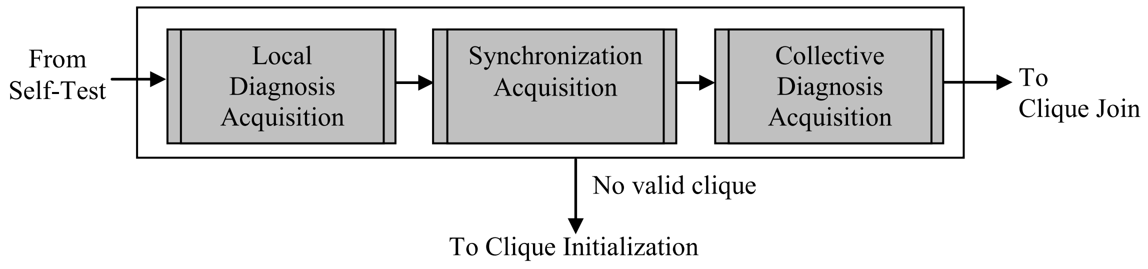

From Self-Test
Local Diagnosis Acquisition
Synchronization Acquisition
Collective Diagnosis Acquisition
To Clique Join
No valid clique
To Clique Initialization

Figure 3.5: Minor modes transitions for Clique Detection mode

3.4.3. Clique Detection

Figure 3.5 shows the minor mode transitions in the Clique Detection major mode. In Local Diagnosis Acquisition, a node uses asynchronous local observations to make a first assessment of the likely members of a clique. In Synchronization Acquisition, the node attempts to synchronize to the clique. In Collective Diagnosis Acquisition, the node captures the health assessment for each node as determined by the clique during the execution of the distributed diagnosis protocol. If at any time during the Clique Detection mode the node determines that a valid clique is not present, it will exit this mode and attempt to form a new clique. Otherwise, it will assume that a clique exists and will try to join it.

3.4.4. Clique Join

When a node enters the Clique Join mode, its state is in agreement with the state of the clique. In this mode, the node runs for two diagnostic cycles, essentially trying to demonstrate that it can be trusted. The existing members of the clique will integrate the node as soon as they confirm that the admission rules have been satisfied.

3.4.5. Clique Initialization

Figure 3.6 shows the minor mode transitions for the Clique Initialization major mode. A node transitions to the Clique Initialization major mode to form a new clique. The first minor mode is Initial Diagnosis, in which a node identifies other nodes that are also attempting to form a new clique. This is followed by the Initial Synchronization and Collective Diagnosis minor modes, where the nodes synchronize their local-time clocks and reach agreement on the clique membership.

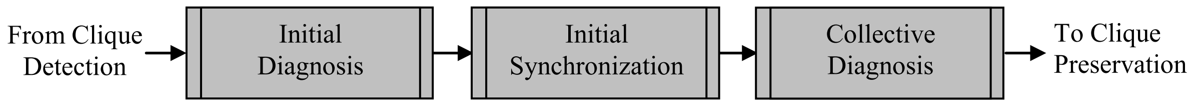


Figure 3.6: Minor modes transitions for Clique Initialization mode

3.5. ROBUS Messages

The BIUs, RMUs, and PEs communicate using ROBUS Messages (RM). Figure 3.7 illustrates the message format, which consists of a Tag field followed by a Payload field. The Tag field has one of two values: SPECIAL or DATA. The format and content of the Payload field depends on the value of the Tag field and the context in which the message is used.

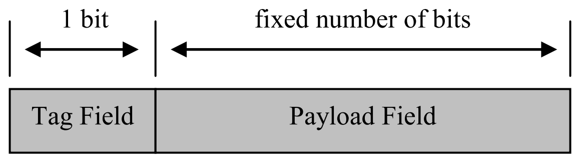


Figure 3.7: ROBUS Message format

A SPECIAL message carries a bit pattern corresponding to one of several labels including, among others, the INIT and ECHO labels used by the synchronization protocols.

DATA messages carry data with a context-specific format. For Collective Diagnosis, the Payload field of each message carries diagnostic information in the form of a Boolean vector. For Schedule Update, a message carries the number of messages scheduled for a particular PE. For the PE Broadcast protocol in the PE Communication mode, the messages carry information from the PEs with an application-dependent format. The exchange of accusations after the completion of the scheduled PE broadcasts uses the payload format for diagnostic messages.

3.6. Point-to-Point Communication

Figure 3.8 illustrates the structure of a one-way communication path between a sending node and a receiving node. The link transmitter and receiver are part of the Communication Module at the source and receiver nodes, respectively. The received messages are stored by the Computation Module at the receiver node until the proper time for processing. This arrangement supports all the modes of point-to-point communication between BIUs and RMUs: synchronous, fixed-delay, and asynchronous-monitoring.

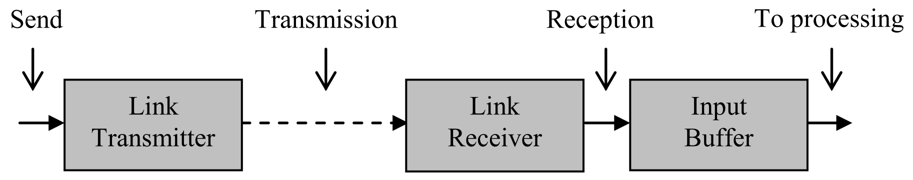


Figure 3.8: Generic point-to-point communication path

Synchronous communication is used with the synchronous protocols. This is a time-triggered communication scheme. Synchronous communication requires that the local-time clocks of the source and receiver nodes be synchronized within some known bounded precision. Figure 3.9 illustrates the main variables. A time T_{REF} is chosen as a reference to coordinate the send and receive actions. A message sent at time T_{SND} with a nominal reception delay R_{PP} is expected to be received at local time $T_{RCV,E}$. Taking into consideration the local-time skew between the source and the receiver, and the uncertainty in the reception delay, a message from a trustworthy source should be received within an expected-reception interval of duration W_{RCV} centered at $T_{RCV,E}$. A message that arrives outside this interval is invalid. A valid received message is buffered until the scheduled time for processing T_{PROC}. This buffering performs a deskewing function in which the received message is synchronized to the local time at the receiving node.

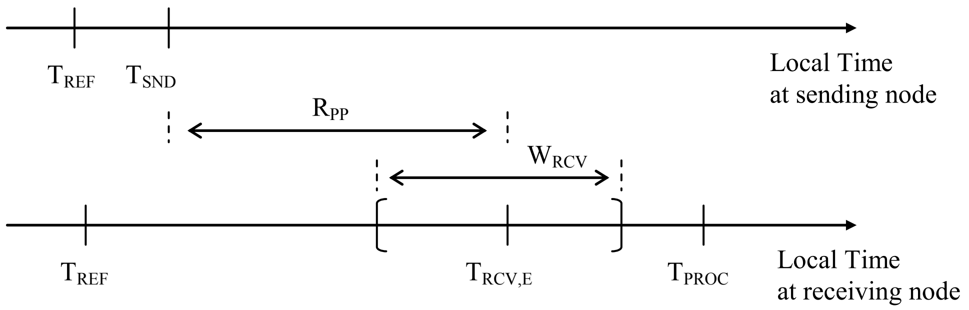


Figure 3.9: Timing for synchronous communication

Fixed-delay communication is used with the synchronization protocols. For this communication mode the information of interest is in the timing of the messages. A transmission is triggered by events at the source. At the receiving end, the message is buffered for a predetermined duration of time before processing it. This communication mode is used only with the INIT and ECHO messages of the synchronization protocols. For the Synchronization Preservation protocol executed in the Clique Preservation and Clique Join modes, it is possible to use local events at the nodes to determine a nominal expected time of reception and an expected-reception interval for the synchronization messages.

Asynchronous-monitoring communication is used by a recovering node to observe the activity on the bus before its local time is synchronized. This communication mode does not require coordination between a source node (which could be a synchronized clique member) and the receiving recovering node. Asynchronous monitoring is made possible by the fact that the BIUs and RMUs broadcast their transmissions to all the nodes of the opposite kind and the point-to-point communication path allows the recovering node to receive messages regardless of its state. The recovering node uses the buffer as a fixed-delay queue, and the messages are processed in the order in which they are received. The delay is not necessarily the same as for the fixed-delay communication mode used with the synchronization protocols.

The PE Interface at the BIUs is designed using a first-in, first-out (FIFO) buffer abstraction for input and output. For input, it is assumed that either a PE message is available when expected or there is a corresponding error indication. For output, it is always assumed that the message can be output at its scheduled time without having to confirm that the PE is ready to receive it.

3.7. Communication Patterns

This section presents the patterns of communication for the ROBUS-2 protocols. The description is limited to the sequences of computation processes and message transmissions. The actual computation operations performed by the nodes are not described here. [Torres05A] has a complete description of the protocols.

3.7.1. Collective Diagnosis

Figure 3.10 shows the communication pattern for the Collective Diagnosis protocol. The circles represent the processing done by the nodes. Each arrow represents a single-message broadcast transmission from the sources to the receivers. BIUs and RMUs use synchronous communication. The results of the protocol are convictions against BIUs and RMUs, which are stored locally by BIUs and RMUs and are also forwarded to the PEs.

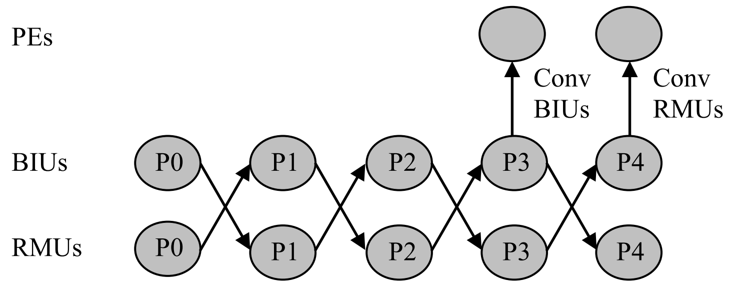


Figure 3.10: Message flow graph for the Collective Diagnosis protocol

3.7.2. Schedule Update

Let N denote the number of BIUs, which is assumed to equal the number of PEs connected to the bus. The PEs are identified according to statically assigned identification numbers which uniquely identify each ROBUS port. The desired schedule is delivered by each PE to its BIU in the form of N consecutive messages with the ordinal positions in the message sequence matching the PE identification numbers in ascending order and the payload fields of each message indicating the desired number of messages to be broadcast by the corresponding PE. The interval between the send time of one message and the send time of the next (known as the data introduction interval or DII) [DeMicheli94] is constant. The submitted schedule messages are processed using an agreement protocol, called the Schedule Update protocol, to ensure that all the BIU and RMU clique members and their PEs agree on the number of scheduled communication messages for each PE. Figure 3.11 shows the message flow graph for the Schedule Update protocol. This is a synchronous-communication protocol. The protocol is applied independently N times, with each iteration processing the messages delivered by the PEs that indicate the number of messages to be broadcast by a particular PE. The result of each protocol iteration is sent to the PEs in process P2. After all the messages have been processed, the ROBUS nodes individually assess the resulting schedule. If the new schedule is valid, it is accepted. Otherwise, a default schedule known to all the PE, BIU, and RMU nodes is used.

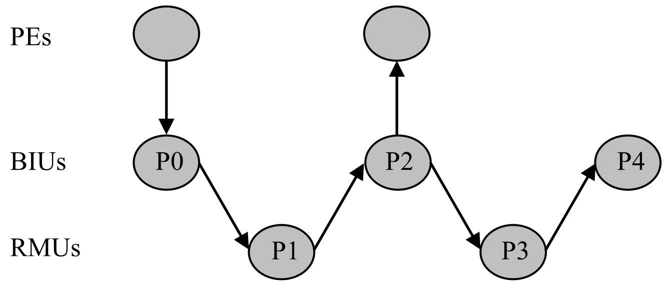


Figure 3.11: Message flow graph for the Schedule Update protocol

3.7.3. PE Broadcast

In PE Communication mode, ROBUS grants bus access to individual PEs according to the communication schedule. An interactive consistency protocol, called the PE Broadcast protocol, is used for each scheduled message to ensure that the PEs receive consistent messages. The bus access pattern is a time-indexed, as-soon-as-possible (ASAP) round-robin sequence. Figure 3.12 provides an example of the access pattern. The PEs access the bus in ascending order according to the port identification numbers. The first scheduled message is sent at some predetermined time. The DII for PE messages is constant. After all the scheduled messages for one PE have been sent, the messages for the next PE are broadcast maintaining the proper DII between messages. If a PE is not scheduled to send messages, then the messages for the next scheduled PE are sent. This continues until all the scheduled messages have been sent.

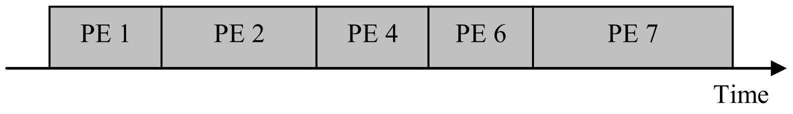


Figure 3.12: Example of an access pattern during the PE Broadcast service

Figure 3.13 shows the message flow pattern for the PE Broadcast protocol. This protocol is used to process each scheduled PE message. Only the scheduled PE and its corresponding BIU are required to send messages. The other PEs and BIUs are allowed to send messages at the same time as the scheduled PE and BIU (for example, in order to reduce node-error detection latency), but those messages will not be forwarded by the RMUs. The result of the protocol is relayed to the PEs. The protocol uses synchronous communication.

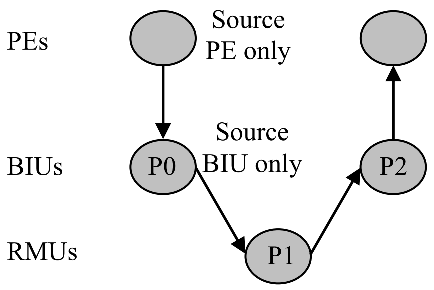


Figure 3.13: Message flow graph for the PE Broadcast protocol

3.7.4. Accusation Exchange

The broadcast of all the scheduled PE messages in the PE Communication mode is followed by the Accusation Exchange protocol, in which the BIU and RMU nodes exchange accumulated accusations against nodes of the opposite kind. Figure 3.14 shows the message flow pattern. This protocol uses synchronous communication.

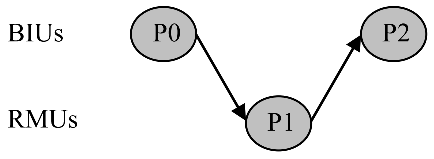


Figure 3.14: Message flow graph for the Accusation Exchange protocol

3.7.5. Synchronization Preservation

Figure 3.15 shows the communication pattern for the Synchronization Preservation protocol. The message to be sent by each process is indicated in the figure. Fixed-delay communication is used for all the messages. For this protocol, it is possible to use the time of transmission of a message in one process to determine an expected time of reception in another process. For example, the RMUs can estimate the expected time of reception for process P3 based on the time of transmission in process P1. [Torres05A] describes in detail how this can be done. A process uses the expected time of reception to perform a timing check on received messages.

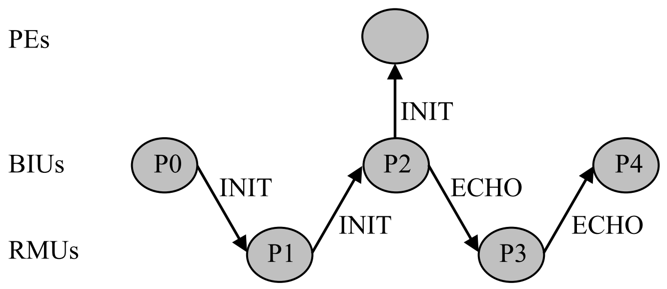


Figure 3.15: Message flow graph for the Synchronization Preservation protocol

3.7.6. Local Diagnosis Acquisition

In Local Diagnosis Acquisition, a recovering node monitors the activity on the bus to determine a trusted set of opposite-kind nodes operating in the Clique Preservation mode. The recovering node uses the asynchronous-monitoring communication mode to make its observations. A node in this mode does not transmit messages. Figure 3.16 illustrates the nominal message flow for a recovering RMU P0 in a 3x3 system (i.e., 3 BIUs and 3 RMUs). The PEs and their links are not shown. The solid arrows represent the message flow from the BIUs to the recovering node. The dashed lines represent the bidirectional communication between the BIUs and the other RMUs. The message flow is similar for a recovering BIU.

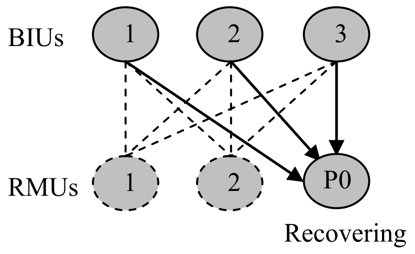


Figure 3.16: Message flow in a 3x3 system with a recovering RMU executing Local Diagnosis Acquisition

3.7.7. Synchronization Acquisition

The Synchronization Acquisition mode has two protocols: Frame Synchronization and Synchronization Capture. The Frame Synchronization protocol monitors the activity on the bus to find the gap between consecutive executions of the Synchronization Preservation protocol. Figure 3.16 also applies to this protocol. The recovering node can use fixed-delay or asynchronous-monitoring communication.

The Synchronization Capture protocol is activated by the end of the Frame Synchronization protocol. A recovering node executing Synchronization Capture receives messages only from the opposite kind nodes and does not generate any messages. In that sense, Figure 3.16 also applies to this protocol. Synchronization Capture uses fixed-delay communication applied to ECHO messages from the Synchronization Preservation protocol. Figure 3.17 shows the message flow graph for the Synchronization Preservation protocol expanded to include the Synchronization Capture processes P3C and P4C. As shown, a recovering RMU executing process P3C receives the ECHO message broadcast by process P2, and a recovering BIU in process P4C receives the ECHO message from process P3. A recovering BIU executing process P4C also sends an ECHO message to its attached PE.

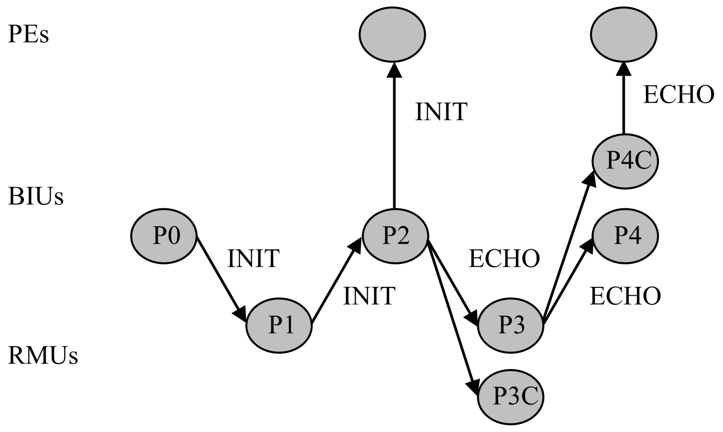


Figure 3.17: Message flow graph for Synchronization Preservation with the Synchronization Capture processes

3.7.8. Collective Diagnosis Acquisition

A recovering node executing the Collective Diagnosis Acquisition protocol is assumed to be synchronized to a clique. The processing in this protocol is essentially the same as for the Collective Diagnosis protocol and is executed by a recovering node in parallel with the execution of the Collective Diagnosis protocol by the clique members. Figure 3.10 shows the message flow pattern for the Collective Diagnosis protocol. A recovering node receives messages using the synchronous communication model. In terms of the communication pattern, the main difference between Collective Diagnosis Acquisition and Collective Diagnosis is that a recovering node does not broadcast messages during the Collective Diagnosis Acquisition protocol. In that sense, Figure 3.16 also applies to this protocol.

3.7.9. Initial Diagnosis

In the Initial Diagnosis minor mode, the nodes execute a synchronous protocol to determine an initial trusted set taking advantage of the known bound on the synchronization precision when operating in the unsynchronized state. Figure 3.18 illustrates the message flow pattern. This protocol uses synchronous communication.

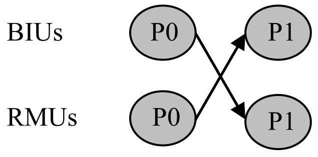


Figure 3.18: Message flow graph for the Initial Diagnosis protocol

3.7.10. Initial Synchronization

The Initial Synchronization protocol is similar to the Synchronization Preservation protocol. The differences in processing are the result of the possibly larger bound on the relative local-time skew at the beginning of the protocol execution. Figure 3.19 shows the message flow pattern. For this protocol, the BIUs send ECHO messages to the PEs from process P4, instead of INIT messages from process P2. This protocol uses fixed-delay communication.

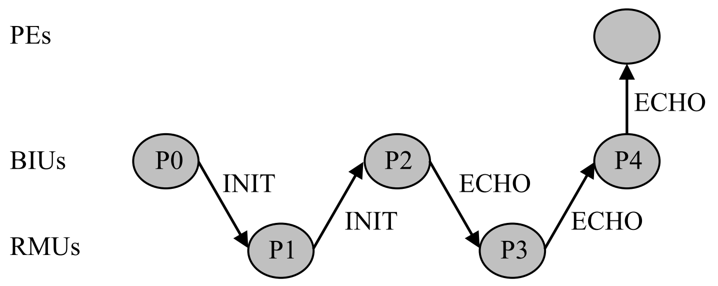


Figure 3.19: Message flow graph for the Initial Synchronization protocol

3.8. Communication between PEs and BIUs

ROBUS requires a bidirectional communication capability between BIUs and their attached PEs. A BIU and its PE can be in separate FCRs or they can share an FCR. If a BIU and its PE are in separate FCRs, the physical communication links must provide adequate barriers to the propagation of faults between the FCRs. In this case, the physical failure of the BIU is independent from the failure of the PE and the failure recovery process of the BIU is completely independent from the PE. Note that if a BIU fails, its attached PE is, in effect, disconnected from the bus. On the other hand, if a BIU and its PE share an FCR, a fault can propagate between the BIU and the PE. In this case, the physical failure of one is no different from a failure of the other. Therefore, the design must provide for the simultaneous recovery of both components. In ROBUS-2, this can be handled by a common process that resets the BIU and the PE when a failure is detected on either of them. Irrespective of the FCR configuration, it is the responsibility of the PE to monitor the communication in order to determine the state of the BIU.

BIUs and PEs exchange messages using two communication models: fixed-delay and synchronous. The fixed-delay model is used with the time synchronization protocols and is essentially the same as for the communication between BIUs and RMUs. The fixed-delay model allows the PEs to synchronize their time using reference events at the BIU interfaces. Synchronous communication is used with the synchronous protocols. Two different synchronous communication models are allowed. In the “tight” model, the transmission of messages between BIUs and PEs follows a strict schedule in which timing is specified down to the tick level. This is the model used for synchronous communication between BIUs and RMUs. In the “loose” synchronous communication model, the sending and receiving of messages by the PEs is only required to satisfy simple timing constraints. Since the BIUs have time-triggered operation, their input and output of PE messages follows a detailed time-indexed schedule. The send timing requirement for the PEs is that their messages must be available at the BIUs at or before the time at which the BIUs will read them. The receive timing requirement for the PEs is that they will get the messages after the BIUs generate them according to their schedule. The specific time at which the PEs send their messages and the delay in receiving BIU messages for the “loose” synchronous communication model is dependent on the implementation and the applications run by the PEs. The PE Interface at the

BIUs is designed using a first-in, first-out (FIFO) buffer abstraction for input and output. For input, it is assumed that each expected PE message is available or there is a corresponding error indication. For output, it is always assumed that the message can be output at its scheduled time without having to confirm that the PE is ready to receive it.

The PEs send messages only during the schedule update and PE Broadcast services. In both cases, only DATA messages are sent. For these messages, the BIUs read the messages and broadcast them on the bus. PE-error checks are not described in this document. These checks are used to signal the BIU when expected messages are not available or otherwise invalid. In either case, a BIU will replace the expected PE message with a SPECIAL message with the payload field set to PE_ERROR.

In addition to service messages, the PEs receive mode and identification messages from the BIUs. The mode messages enable a PE to track the mode of its BIU. A BIU will send a mode message to its PE every time there is a major mode transition and after every diagnostic cycle during the Clique Join and Clique Preservation modes. The mode messages are SPECIAL messages with the payload field set to SELF_TEST, CLIQUE_DETECTION, CLIQUE_INITIALIZATION, CLIQUE_JOIN or CLIQUE_PRESERVATION, as appropriate. The ID message carries the BIU's identification number, which is the identification number to be used also by the PE. These are DATA messages with the payload field equal to the BIU's identification number. This way of giving an identification number to a PE is preferred over setting it directly at the PE because it allows the use of generic software at the PEs and prevents a mismatch between the BIU and the PE identification numbers.

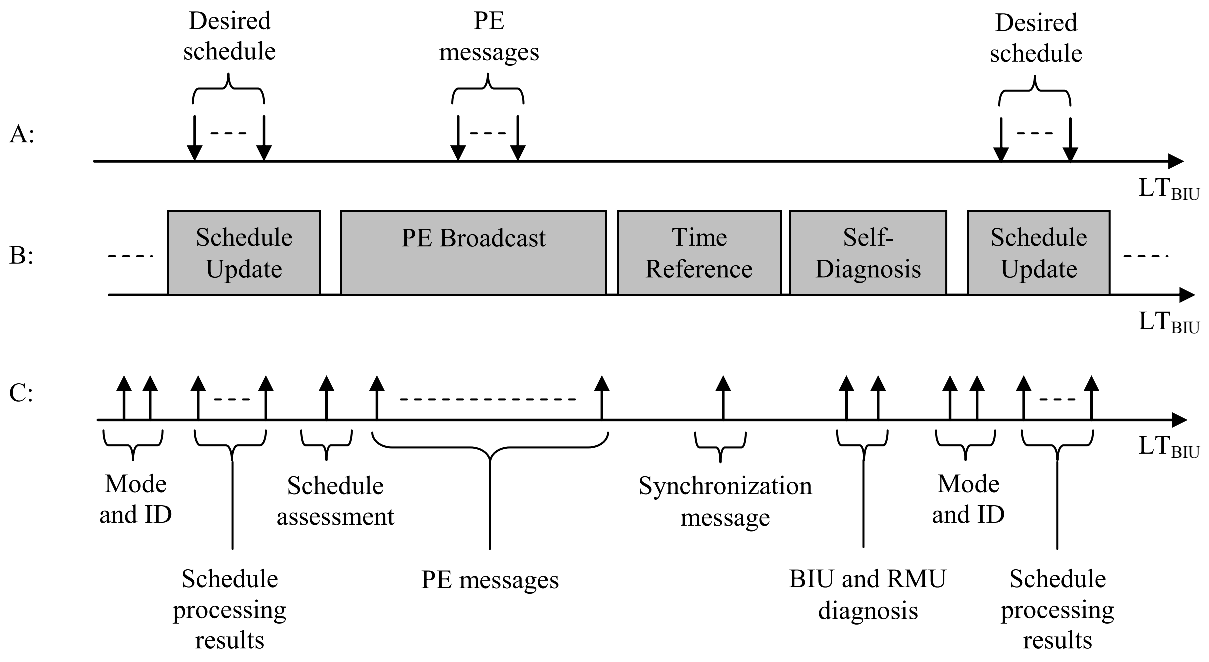


Figure 3.20: Message exchange pattern between a BIU and its attached PE in Clique Preservation mode

A: PE-to-BIU messages; B: Bus services; C: BIU-to-PE messages

Figure 3.20 illustrates the message exchange between a BIU and its attached PE during the Clique Join and the Clique Preservation modes. The mode and identification messages are sent to the PE between the Self-Diagnosis and Schedule Update services. During Schedule Update, the BIU reads the schedule submitted by its PE and sends to the PE the results determined by the bus for each PE. This is followed by a single message with the assessment result for the new schedule. This consists of a SPECIAL message with the payload set to VALID_SCHEDULE, ZERO_SCHEDULE (i.e., the schedule is valid and equal to zero for all the PEs), or INVALID_SCHEDULE. If the schedule is invalid, ROBUS will automatically switch to a default schedule. During the PE Broadcast service, a BIU will read the scheduled messages from its PE and output to the PE the result for all the scheduled messages. A broadcast result equal to PE_ERROR indicates that there was an error at the source PE. If the bus determines that the BIU of a source PE is not operating properly, then the result of the broadcast will be SOURCE_ERROR or NO_MAJORITY. A result of NO_MAJORITY indicates that the RMUs received different messages from the source BIU. If the assessment of the schedule was ZERO_SCHEDULE, the PE Broadcast service is not executed and ROBUS simply waits until it is time to execute the Time Reference service. During the Time Reference service, a BIU outputs a SPECIAL message with the payload set to INIT. The sending of this message is triggered by the reference event that the BIU will use to reset its local-time clock. For Initial Synchronization and Synchronization Acquisition, the payload is set to ECHO to explicitly indicate that a different protocol event is used as a reference to reset the local-time clock. During the Self-Diagnosis service, the output of a BIU consists of two messages containing the diagnostic results for the BIUs and the RMUs. These messages complete the service cycle. The next cycle begins with the mode and identification messages.

4. Overview of the Configurable Fault-Injection and Monitoring System

The system described in this section is the first step in the capability development to serve the NASA-ODU research needs. We give a high level description of the initial CFIMS realization based on the concept presented in Section 2. The first two subsections here are overviews of the application and the execution platform, as they are strong determining factors for many SUT and CFIMS design and implementation decisions. The description of the CFIMS and the relevant aspects of the SUT are presented after that. A more detailed description of all system aspects is given in the remaining sections of this report.

4.1. Initial Application

The goal in the first set of experiments is to characterize the application-level effects of faults in the ROBUS-2 communication system. For this, ROBUS-2 will be exposed to physical and simulated faults while data is collected on the effects on PE messages carrying application data. Then, the experimental communication-fault effects data will be used in closed-loop simulations of a Boeing 747 airplane controlled by an autopilot to measure the fault effects on the performance of the closed-loop system. The expected results of this work are validated models of fault effects on the ROBUS-2 communication service and the performance of the closed-loop system.

In essence, the experimental setup consists of a set of PEs exchanging messages according to a predetermined communication schedule while faults are injected into ROBUS and observations are collected about the results of expected message receptions at the PEs. The PEs will not be programmed for dynamical closed-loop control of a plant. Instead, they exchange static messages of appropriate length for sensor readings and effector commands with a pattern of communication over the bus that is representative of an actual control application. The PEs execute their operations repetitively with a predetermined constant execution period. An execution iteration is referred to as a control cycle. The PEs will be grouped and labeled according to the functions they would perform in an actual control system, with some PEs executing input and output application processes and others executing the control process, as illustrated in Figure 4.1. This arrangement with a group of PEs running both input and output processes (see IO PEs in Figure 4.1) reduces the total number of PEs required in the system and allows the use of the existing ROBUS Protocol Processor design [R2PP], which is not intended for large systems. In every control cycle, each IO PE transmits on the bus one set of sensor readings and each Control PE transmits one set of commands. An application-level PE message maps to one or more ROBUS messages. A control cycle consists of one or more ROBUS cycles.

Other than application data communication, none of the standard PE-level SPIDER services described in Section 1 will be implemented in the first configuration. The operation of the PEs will be coordinated by an external centralized process responsible for providing control cycle-level synchronization. Future system configurations may implement internal distributed coordination protocols for PE-level diagnosis, synchronization and redundancy management.

Although the intended fault injection targets for the initial experiments are the ROBUS components, namely BIUs and RMUs and their communication links, the minimum units of failure in a fault tolerance and redundancy management sense are actually defined in terms of fault and error containment regions (FCRs and ECRs). If a PE and a BIU share an FCR, any fault affecting the BIU makes the PE untrustworthy and both are considered to fail as a unit. Even if a PE and its BIU do not share an FCR, the PE still does not fail independently of its BIU because a PE requires a properly operating BIU for access to the bus, without which a PE becomes untrustworthy as it has no means to coordinate its operations with the rest of the system. In effect, the failure of a BIU propagates and corrupts the PE. Thus, given that the BIU is a fault injection target, a PE-BIU pair must be treated as a single unit of failure.

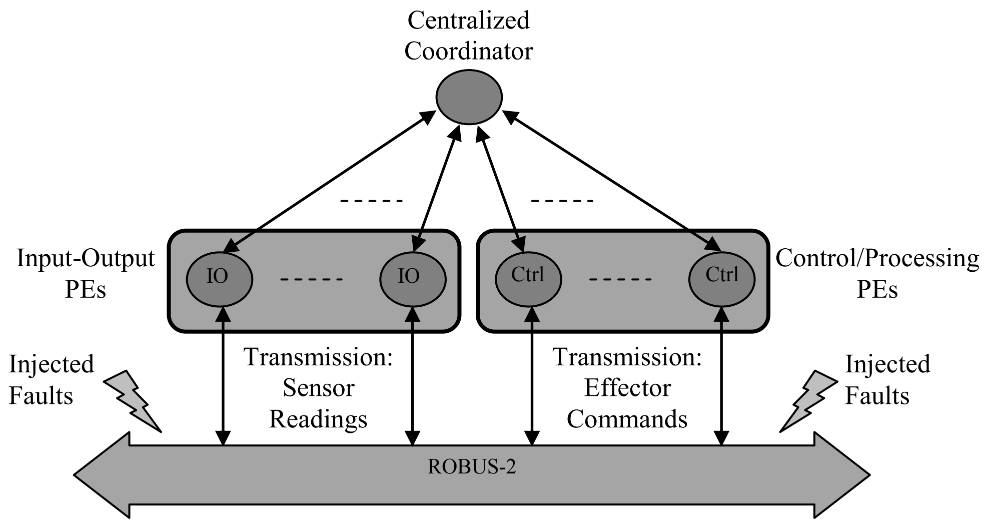


Figure 4.1: Concept for initial SUT configuration

4.2. Execution Platform

In a full SPIDER system, the RPPs for BIUs and RMUs have custom hardware designs [Torres05B], and the PEs have hardware- and software-implemented functions. The execution platform selected for this project enables the implementation of distributed computation networks and consists of general purpose reconfigurable processing nodes with hardware and software programmable resources and basic communication resources. The execution platform is used to implement the SPIDER-based SUT and the CFIMS. As such, the execution platform must support simulated fault injection experiments and physical fault injections in HIRF and neutron radiation environments.

This subsection describes the execution platforms for this project. The first platform, RSPP1, where RSPP stands for Reconfigurable SPIDER Prototyping Platform, is an existing platform acquired from a previous project. The second platform, RSPP2, is an upgraded version with increased performance and higher damage tolerance in neutron radiation environments.

4.2.1. Reconfigurable SPIDER Prototyping Platform 1

The RSPP1 was developed for NASA by Derivation Systems, Inc (DSI) under a phase III SBIR contract and consists entirely of commercial off-the-shelf (COTS) parts. The RSPP1 is a Field Programmable Gate Array (FPGA)-based development system for the design and testing of SPIDER prototypes. The architecture is a scalable modular system composed of individual RSPP1 nodes interconnected with point-to-point fiber optic links. Figure 4.2 shows two views of an RSPP1 node. An RSPP1 node is a PC/104-plus [PC/104] computer system with the following functional hardware components.

- PF3100-2V3000 PC/104+ FPGA Module: PF3100 with the Xilinx Virtex-II XC2V3000 (3 million gates) FPGA.

- PFBR104 PC/104 Fiber Optic Transceiver Module: Interfaces with the PF3100 over the PF3100 IO connector and provides four Agilent HFBR-5905 fiber optic transceivers. Each RSPP1 node has two PFBR104 modules providing a total of 8 fiber optic IO channels.

- PC/104+ CPU Module: The PC/104+ CPU module is a Lippert CRR2 with a 300 MHz Pentium class processor, 64MB SDRAM, 256MB Compact Flash, 10/100 Ethernet, VGA, Keyboard, RS232 serial port, parallel port, USB port, and cables. The White Dwarf Linux operating system is installed on the CPU module.

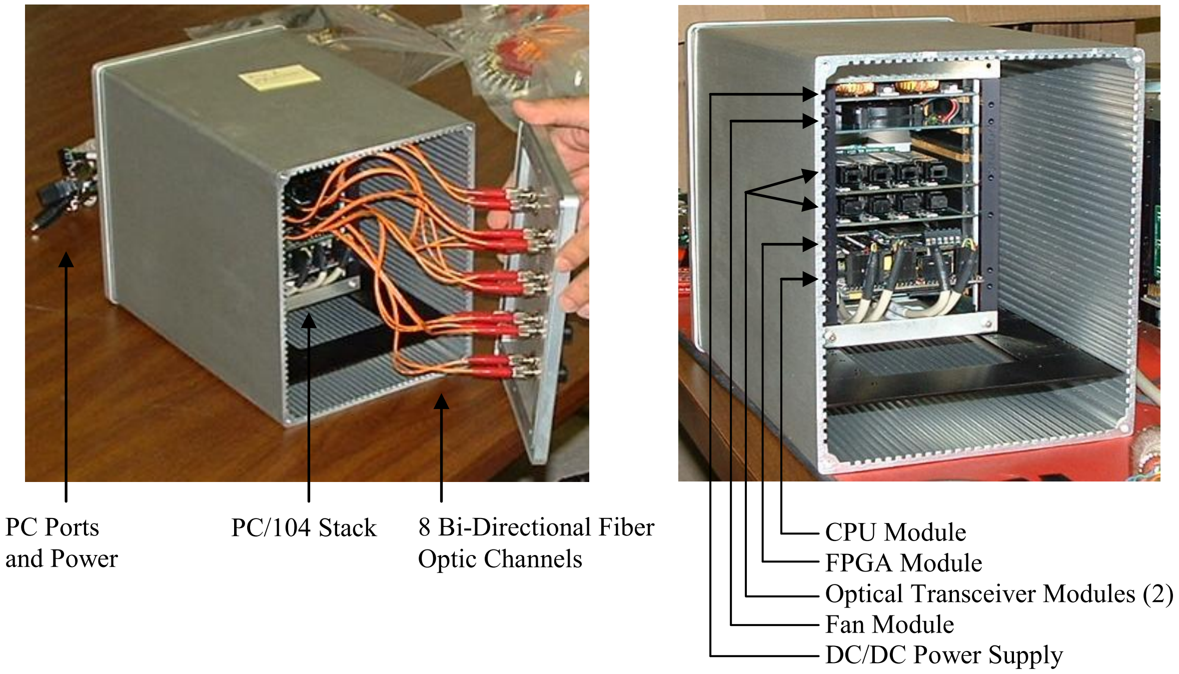


Figure 4.2: RSPP1 Node

Figure 4.3 shows the interconnection of the main physical components in an RSPP1 node. The custom hardware functions of a node are implemented as hardware processes running on the static random-access memory (SRAM)-based FPGA. The program for the FPGA is stored in non-volatile flash memory together with functional parameters for the hardware processes. The FPGA is connected via low-voltage differential signaling (LVDS) [XAPP230] lines to a group of eight fiber-optic transceivers, each composed of independent transmit (Tx) and receive (Rx) sections used for point-to-point communication with other nodes. The complex programmable logic device (CPLD) controls the loading of the FPGA from flash when triggered by the CPU-controlled reset signal on the ISA bus. The CPLD acts as a pass-through for reset requests generated by the FPGA. This node-wide reset feature supports a policy in the ROBUS-2 design that requires a node to reset itself immediately upon the detection of an internal error or a bus failure [Torres05A].

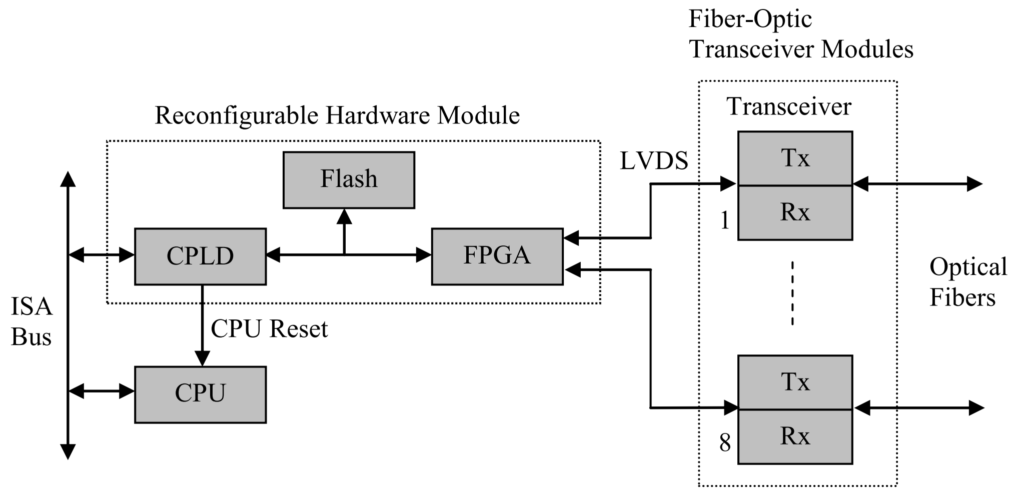


Figure 4.3: Interconnection of the main physical design components on an RSPP1 Node

4.2.2. Reconfigurable SPIDER Prototyping Platform 2

The RSPP1 has several disadvantages in the operating system, the FPGA and the fiber optic transceivers. The RSPP1 operating system, White Dwarf Linux, is a version of Linux intended for embedded applications with a small memory footprint. This version of Linux does not provide support for hard-real-time software execution, and that makes it unsuitable for PE-level software functions in a time-triggered system like SPIDER.

The SRAM-based FPGA on RSPP1 has the drawbacks that it loses its programming every time there is a hardware reset, and there is a long delay in loading the program. This programming delay has a severe impact on the re-initialization delay of a ROBUS-2 system with RPPs running on these FPGAs. The communication system in a flight controller must have very short fault recovery delay to enable fast system recovery for preservation of closed-loop system stability. (Note that the long recovery delay of a ROBUS-2 system using RSPP1 is not a problem in the experiments because no actual controller is being implemented in the initial SUT configuration and what we are trying to measure is the failure statistics without the temporal component in a typical communication exchange.)

The RSPP1 fiber optic transceivers are standard commercial-grade communication parts that have high functional sensitivity and low damage tolerance in high-energy particle radiation environments. If a SPIDER system executing on RSPP1 is exposed to such environments, there is a high probability that most observed faults will be due to fiber optic transceiver faults and that the transceivers will only operate for a short time before being permanently damaged by the radiation.

In the RSPP2 platform, which will be developed in-house at LaRC, the CPU module will be replaced by one with a more recent processor and a real-time operating system. A new reconfigurable hardware module eliminates the CPLD and replaces the SRAM-based FPGA with a flash-based FPGA that does not lose its programming after a power cycle and does not need to be reprogrammed after reset. The fiber optic communication module will be replaced by an electrical communication module that has higher tolerance to particle radiation. In addition, a latch-up protection circuit will be added to the PC/104 stack to reduce the likelihood of permanent damage due to the radiation. [Layton98] describes the performance of one such latch-up protection circuit.

4.3. Initial Configurable Fault Injection and Monitoring System

Figure 4.4 illustrates the architecture of the SUT and the CFIMS. As described in Section 2, the CFIMS enables the collection of observations about the response of the SUT when operating under conditions determined by the configuration, workload and faultload specifications. The system functions are SUT function testing, fault injection, state monitoring, execution control, data collection, and system configuration. These functions are implemented as a set of coordinated distributed processes running on test control nodes and the SUT.

For the given initial application, SPIDER consists of PE-BIU and RMU functional nodes running on separate RSPP physical nodes, with each node forming an independent fault and error containment region. Each PE-BIU functional pair executes on the same RSPP node because of the failure interdependence between the PE and the BIU. That arrangement is also the most representative of an actual fielded system. Each SPIDER node includes local processes for fault injection and state monitoring to support the CFIMS functions. The runtime configuration capability of the nodes is embedded in their local processes.

As shown in Figure 4.4, the main CFIMS functions reside in the test controllers. This distributed multi-controller arrangement satisfies a design requirement of at least one dedicated test communication link for each SPIDER node in SUT configurations as large as 12 BIUs by 4 RMUs, which is believed to be the largest system that may be of interest relative to the NASA-ODU research goals. That connectivity requirement cannot be supported by a controller running on a single RSPP node with only 8 customizable point-to-point communication ports. The chosen control architecture consists of one primary controller connected to as many secondary controllers as needed to reach each SPIDER node with a dedicated test link. The configuration in Figure 4.4 is intended for SUTs with at most seven PEs due to number of available communication ports on an RSPP node.

In the Primary and Secondary Test Controllers (PTC and STC), the functional processes are allocated to either hardware or software taking into consideration the performance requirements of the design, as well as the strengths and weaknesses of each computation paradigm. The following are among the critical design requirements for the test controllers.

- Referencing of distributed observations to a common timeline
- Time-triggered operation for interaction with SPIDER
- High error detection coverage in the communication between the controllers and SPIDER
- Support very high rate of execution on ROBUS
- High fault injection controllability and repeatability

All these requirements are best satisfied by exploiting the high degree of distributed time synchronization precision and operation coordination achievable with hardware implemented functions. However, hardware implementations tend to be limited by available resources and time-consuming to design and analyze due to high execution parallelism and complex interactions between functional components. Software implementations tend to have large available resources (e.g., memory) and be less time consuming due to the sequential-programming model, but also tend to allow only coarse timing control and coordination (relative to what can be achieved with hardware) between parallel processes.

The process allocation shown in Figure 4.4 has a mix of hardware and software functionality that offers a reasonable balance between performance and design complexity.

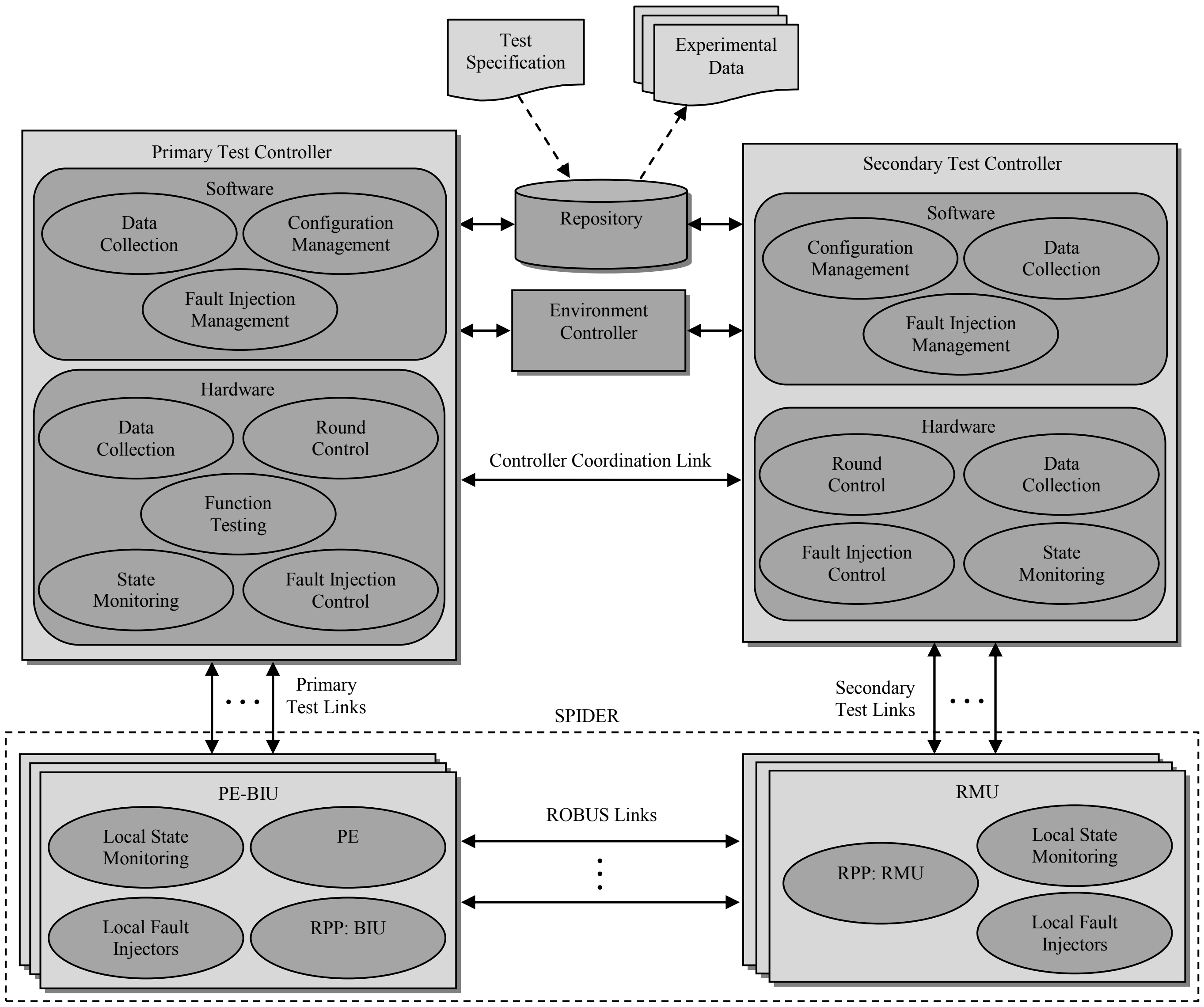


Figure 4.4: High-level architectural view of the CFIMS and SUT

The Test Control Links (TCLs) are a communication infrastructure shared by the CFIMS hardware processes. The TCLs include the Controller Coordination Link (CCL) between the Primary Test Controller and the Secondary Test Controller, the Primary Test Links (PTLs) between the PTC and the PE-BIU nodes, and the Secondary Test Links (STLs) between the STC and the RMU nodes. The TCLs use custom-designed data communication units with high-precision point-to-point communication delay and very low bit-error rate (see Section 6). The Test Control Messages (TCMs) communicated over these links have a common format which is described in Section 7.

At the PTC and STC, the Round Control hardware processes are responsible for overall coordination of activities at the test controllers. A (test) round is the sequence of actions by the system to perform the work described in a Test Specification file (see Section 13 and Appendix A). The Round Controllers execute a Controller Coordination Protocol (CCP) over the CCL to synchronize and coordinate their actions (see Sections 4.3.4 and 5). This protocol leverages the SUT function-based time reference generated by the Function Testing process at the PTC to enable the Round Controllers to provide the common global time service needed to correlate the activities and observations in separate parts of the system for the duration of the round.

The Function Testing process at the PTC has two main components: a function timer, and a function tester and monitor. The function timer is the centralized PE coordinator introduced in Section 4.1. Its purpose is to enable the PEs to synchronize their application level activities and to provide a precise SUT-based time reference with which to coordinate the CFIMS operations. The function tester and monitor component generates the application data to be exchanged by the PEs over ROBUS and gathers fault effects observations based on the messages sent back by the PEs. Section 4.3.1 gives a more detailed description of the Function Testing process.

The Fault Injection processes work together to generate a physical or simulated faultload according to the Test Specification. For physical fault injection, the PTC and STC interact with the Environment Controller to coordinate the activation of environmental disturbances with the operation and monitoring of the SUT. The simulated fault injection capability is intended to be highly flexible and configurable to achieve the desired trade-off between complexity of design and effectiveness of fault injection relative to the properties presented in Section 2.2 (i.e., reachability, controllability, repeatability, reproducibility, and non-intrusiveness). The simulated fault injection processes can be present in both hardware and software in every node of the system, and their individual actions can range from independent decisions based on locally available system state information to globally coordinated actions using the high-bandwidth communication infrastructure. The actions of the fault injection processes can be event-triggered based on monitoring of local or global conditions, or the actions can be driven by time using the available SUT-derived function time. Random fault injection is also possible using the available resources. More details about the fault injection capability are given in Section 4.3.2.

For State Monitoring, every SPIDER node has a local process that reports state observations through the TCLs to higher level monitoring processes. The embedded monitoring processes can be configured for the triggering conditions and the variables to be sampled. In addition to forwarding these observations to the data collection software processes, the state observations can be combined with observations about the SUT application to ascertain in real time the general health status of the SUT.

The Data Collection processes at the test controllers gather observations about the operation of the SUT and the CFIMS throughout the round. The collected data is organized into separate buffers and transferred at the end of the round to files at the Repository, where they remain available for post-test analyses.

The software Configuration Management process is responsible for setting up the system according to the description given in the Test Specification. The configuration information is applied directly at the test controllers, or it can be routed through the system using TCMs to reach particular destination processes. Each receiving process must then apply its configuration information as appropriate in preparation for the next execution round.

The following subsections provide additional information about the design of the SUT and the CFIMS.

4.3.1. SUT Function Testing

As stated previously, the PEs in the initial version of the SUT will not implement any of the standard PE-level distributed coordination and resource management protocols over ROBUS (for example, synchronization, diagnosis and redundancy management), and instead will use an external time synchronization agent to enable coordinated message-passing on the bus. In this subsection, we give an overview of the PE synchronization protocol and the function to exercise and monitor the communication of the PEs over ROBUS.

4.3.1.1. PE Synchronization

The initial SUT application consists of the PEs executing a pattern of communication over the bus that is representative of actual control applications with messages for sensor readings and effector commands. On SPIDER, the coordination of distributed operations is based on the fault-tolerant time service provided by ROBUS in the form of a constant-period cyclic time reference. For a control application with periodicity requirements on the updates of sensor readings and effector commands, the duration of a control cycle would normally be specified as a multiple of the ROBUS cycle. The actual durations of the ROBUS cycle and the control cycle are selected considering a variety of factors like the required synchronization precision, available communication bandwidth, fault arrival rate, fault recovery delay, and control-cycle period and period jitter.

The PE communication pattern over the duration of a control cycle can be optimized by changing the PE communication schedule in every ROBUS cycle using the ROBUS dynamic schedule update service. To do this, the PE communication schedule for a control cycle is defined as an indexed set of ROBUS communication schedules, with one ROBUS schedule for each ROBUS cycle in a control cycle, and with the schedule index identifying the ROBUS cycle in which to apply a particular ROBUS schedule. Accordingly, for proper coordinated communication, the PEs must have agreement on the current time, which includes the ROBUS Time (RT) (i.e., the time within a ROBUS cycle) and the ROBUS Cycle Index (RCI) (i.e., the identifying sequence number for a ROBUS cycle within the control cycle). The ROBUS time service ensures agreement on the RT among the PEs. In the first SUT implementation, RCI agreement among the PEs is achieved by means of a centralized agreement generation and preservation process residing at the PTC and referred to as the Function Timer (FTmr). In addition to providing RCI agreement for the PEs, the FTmr maintains a Control Cycle Index (CCI) to identify and count the control cycles since the beginning of SUT application execution in the current round. The PE RCI agreement protocol described below enables the FTmr to provide all CFIMS components with an SUT-referenced time service consisting of the 3-tuple (RT, RCI, CCI).

Figure 4.5 shows the transition graph for the PE major modes. The OCL, which stands for Operation Coordination Level, measures the degree of readiness of a PE to participate in distributed SPIDER operations. After a reset, in OCL0 a PE must first synchronize to the ROBUS Time as indicated by its attached BIU. Then, in OCL1 the PE waits to confirm that its BIU has reached steady-state in the Clique Preservation Mode (CPM) (see Section 3). Once the PE has stable access to the bus in OCL2, the next step is to synchronize the local RCI and transition to OCL3 at the next control cycle boundary, which is identified by an FTmr synchronization message with RCI set to 0. A PE in OCL3 is ready for normal control-cycle-level communication on the bus. Whenever the PE detects a failure condition, it immediately halts all internal operations and remains there until an external reset is applied.

Figure 4.6 shows the message-flow graph for the RCI agreement protocol. A PE participates in this protocol only after it has reached OCL2, by which time its local RT is synchronized by INIT messages from the attached BIU executing the ROBUS Synchronization Preservation protocol. In process P2IO, a PE transmits a synchronization message after a predetermined constant delay from the time it received the INIT message. The PE message contains the local RCI value. Given known bounds on the point-to-point communication delay, the FTmr can use the time of reception of the message in process P3IO to estimate the RT at the sending PE. In the initial version of the protocol, the FTmr discards the received RCI values and performs a middle-value-select event vote on the time of reception of PE messages. The FTmr then transmits a message a fixed delay after the event vote produces its output. This message carries internally generated values for the RCI and CCI. In process P4IO, the PEs accept the received RCI and use it as the communication schedule index in the next execution of the ROBUS schedule update protocol. The FTmr message (referred to as the "sync message") is also distributed throughout the CFIMS to enable SUT-referenced time-coordinated distributed actions.

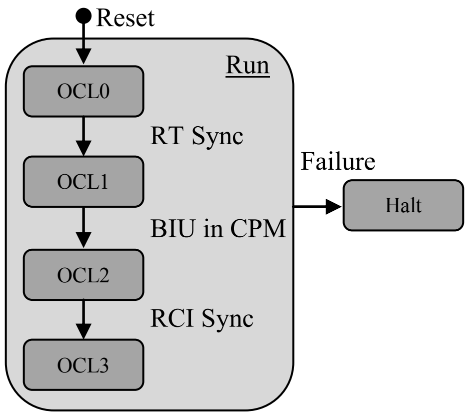


Figure 4.5: Major-mode transition graph for a Processing Element

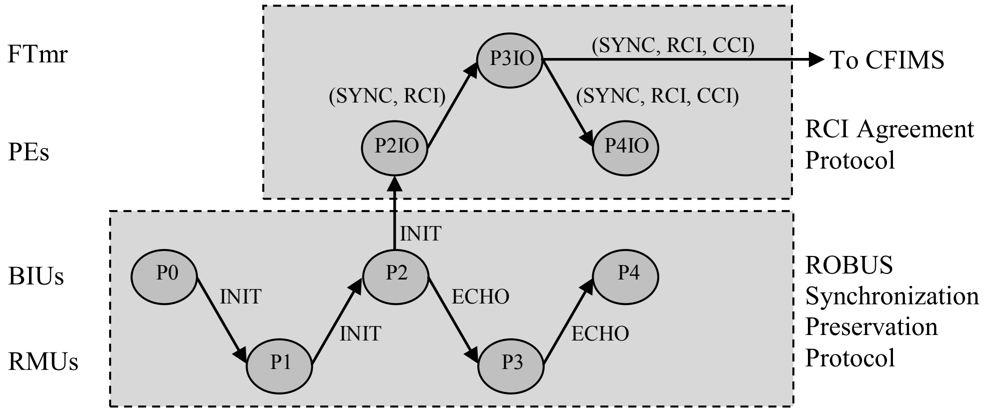


Figure 4.6: Message flow graph for the RCI agreement protocol

The design of the FTmr uses many synchronization concepts from ROBUS and the RPP. Figure 4.7 shows the FTmr mode transition graph. After a reset, the FTmr operates asynchronously with respect to SPIDER. It first enters a passive phase (i.e., no output generation) of monitoring and diagnosing the synchronization messages from the PEs in order to establish an initial set of trusted PEs. This minor mode is similar to the RPP's Local Diagnosis Acquisition in the Clique Detection major mode. When the FTmr determines an initial valid set of PEs, it transitions to RT Frame Synchronization mode where it finds the gap between consecutive bursts of synchronization messages from the PEs. Notice that the PEs send their synchronization messages from process P2IO in Figure 4.6 as a result of receiving synchronized INIT messages from the BIUs, which are generated periodically when ROBUS executes its Synchronization Preservation protocol. The FTmr Frame Synchronization protocol ensures that RT Synchronization Capture will not start while a burst of synchronization messages from trustworthy PEs is being received. The FTmr synchronizes to the current RT in the Synchronization Capture mode. When RT synchronization is achieved, the FTmr transitions to RT Synchronization Preservation and begins SUT-synchronous operation. Whenever a failure condition is detected, the FTmr restarts the synchronization process in the Initialization major mode.

Additional information about the RCI agreement protocol can be found in Section 9 for the PEs and in Section 11 for the FTmr.

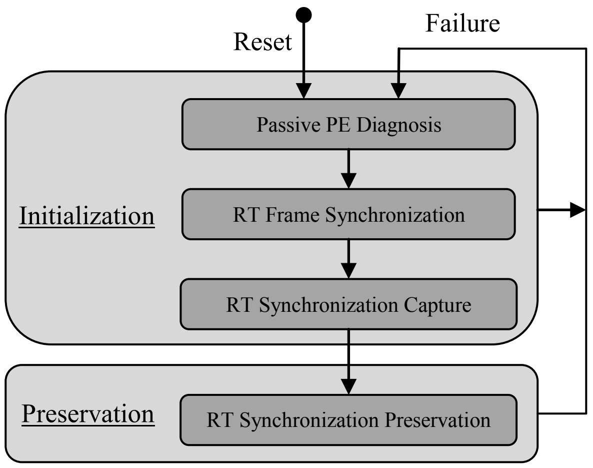


Figure 4.7: Mode transitions for Function Timer

This version of the RCI agreement protocol satisfies the intent of having a single agent external to SPIDER that generates the RCI values to ensure agreement among the PEs. However, discarding the received RCI values at the FTmr introduces an unnecessary degree of artificiality in the behavior of the system due to the fact that there is no relation between the RCI value generated by the FTmr and the RCI values at the PEs, which are part of the system state at the PEs. It is a simple exercise to modify this protocol to compute the RCI at the FTmr using a majority word vote over the received values from the PEs. That modified version of the protocol can be used to fully synchronize the FTmr as a slave process to SPIDER in future versions of the system in which the PEs execute a distributed agreement protocol by exchanging their local RCIs over ROBUS. In that system, process P4IO at the PEs is redundant and can be removed.

4.3.1.2. Application Testing

In the first SUT application, the PEs exchange messages over ROBUS while observations are collected about fault effects. The PEs are grouped into IO PEs that transmit sensor reading messages and Processing (or Control) PEs that transmit effector command messages. Every PE is allowed to transmit exactly one application-level message per control cycle. The application communication schedule is predetermined and repeated every control cycle. The assignment of applications to the PEs as either IO or Processing is given in the Test Specification (see Figure 4.4). The Test Specification also indicates which PEs will be active during the test round.

Continuing with the model of having a centralized PE coordinator, in every control cycle the PTC Function Testing process sends to the PEs the application data to be transmitted on ROBUS and generates observations based on the communication results reported by the PEs. Essentially, the goal is to check the communication paths between PEs. To do this, the PTC Function Testing process has a dedicated lane Function Monitor (FMon) for each PE. Figure 4.8 illustrates the message flow from FMon_i to FMon_j. The FMons read the current system time from the FTmr. At the beginning of a control cycle, FMon_i sends a message to PE_i containing either sensor readings or effector commands, depending on whether PE_i is configured for IO or Processing, respectively. If PE_i detects an error in the message from FMon_i, it makes a record of the error and sets the appropriate field in the message content to be transmitted on ROBUS, assuming PE_i is scheduled to transmit. PE_i broadcasts a message on ROBUS at the scheduled time, and the message is received by every active PE, including PE_i itself. If PE_j detects an error in the message received on ROBUS, it makes a record of it. When PE_j is ready, it adds the scheduled source Id to the message (i.e., the Id for PE_i) and an indication in the appropriate message field if a message reception error was detected, and then sends the message over to FMon_j. FMon_j inspects the received message and generates an observation based on the content. At the end of every control cycle, FMon_j outputs one observation result for each PE as a source, irrespective of whether the PE actually was a message source during the cycle.

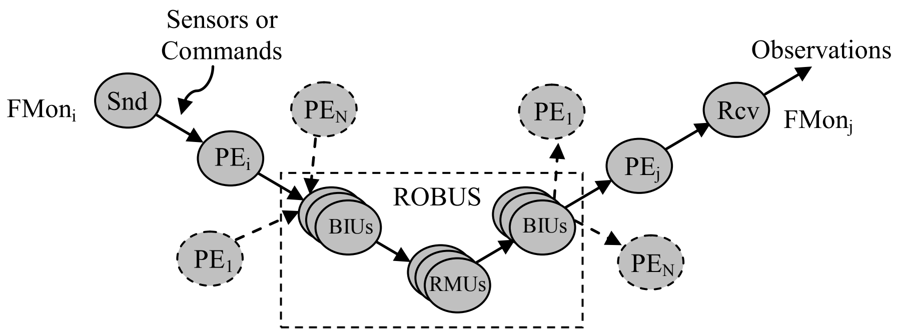


Figure 4.8: Message flow pattern for testing ROBUS fault effects on PE messages

Table 4.1 lists the possible observation results for the path from FMon_i to FMon_j. To generate this table, it is assumed that there are no faults at the FMons, there are no restrictions in how the PEs, BIUs, or RMUs may fail (i.e., they may experience omissive or transmissive faults, with transmissive faults having a non-negligible probability), and there is near-perfect integrity in the communication over the PTLs and RLs (i.e., either a message receiver gets the correct message or it detects a bad message) such that transmissive communication faults are possible but unlikely. These assumptions are supported by the fact that fault injection does not target the FMons directly; the implementation of the PE, BIU, and RMU nodes may not have local-error detection mechanisms with high detection coverage; and the point-to-point communication links to be used are designed for high integrity (see Section 6). Every FMon handles received messages from its corresponding PE in an event-triggered manner (i.e., messages are immediately processed when they arrive) and with a control-cycle time granularity for the generation of observations (i.e., messages are processed at their actual time of arrival during the control cycle, but all the observations are outputted at the end of the control cycle). If an FMon does not receive a message with a particular Sender Id during the control cycle, then the observation result for that Sender Id at the end of the cycle will be the default one (i.e., Omitted Sender Id). When the FMon receives a message, the error checks are applied in the priority order indicated in Table 4.1. The outputted observation for a message will be determined by the first check to detect an error. A message is considered good (i.e., correct) only if it passes all the checks. Note that transmissive faults are considered more serious (and interesting) from an effects characterization standpoint.

Table 4.1: Defined observations for the communication path from sender $FMon_i$ to receiver $FMon_j$

Observation by $FMon_j$	**Check Priority**	**Check Description**	**Explanation of Observation at $FMon_j$**
Omitted Sender Id	0	$FMon_j$ received no message from $FMon_i$.	- Default observation applicable to the case of PE_i disabled or not scheduled to transmit. - May also be due to a fault in PE_j or the PTL from PE_j to $FMon_j$.
Invalid Sender Id	1	$FMon_j$ received a message from $FMon_i$ in a system configuration in which PE_i (and thus, $FMon_i$) does not exist or is not active.	- Mostly likely due to a transmissive fault in PE_j. - May also be due to a transmissive fault in the PTL from PE_j to $FMon_j$. - The check for this observation is given higher priority than the one for “Bad Payload Length” because the outputted observations are indexed by the sender Id. It is also given higher priority than “Repeated Sender Id” because it could be interpreted as an error due to a fault in PE_j.
Repeated Sender Id	2	$FMon_j$ received multiple messages from $FMon_i$.	- Most likely due to a transmissive fault in PE_j. - May also be due to a transmissive fault in the PTL from PE_j to $FMon_j$. - There is no basis to give priority and check the content of any one of multiple messages with the same sender id. - The check for this observation is given higher priority than the one for “Bad Payload Length” because the outputted observations are indexed by the sender Id.
Bad Payload Length	3	$FMon_j$ received a message from $FMon_i$ with an incorrect message length for the application assignment given to PE_i.	- Most likely due to a transmissive fault in PE_j. - May also be due to a transmissive fault in the PTL from PE_j to $FMon_j$. - The correct message length is determined based on the application assignment for the PE with the received sender Id.
Detected Reception Error at Receiver PE	4	$FMon_j$ received a message from $FMon_i$ with a reception error reported by PE_j.	- Most likely due to an omissive fault in PE_i or ROBUS. - May also be due to a transmissive fault in PE_j or the PTL from PE_j to $FMon_j$.
Detected Reception Error at Sender PE	5	$FMon_j$ received a message from $FMon_i$ with a reception error reported by PE_i.	- Most likely due to an omissive fault in the PTL from $FMon_i$ to PE_i. - May also be due to a transmissive fault in PE_i, ROBUS, PE_j or the PTL from PE_j to $FMon_j$.
Bad Message Content	6	$FMon_j$ received a message from $FMon_i$ with an incorrect message content for the application assignment given to PE_i.	- Most likely due to a transmissive fault in PE_i or PE_j. - May also be due to a transmissive fault in the PTL from $FMon_i$ to PE_i, ROBUS or the PTL from PE_j to $FMon_j$. - The correct message content is determined based on the application assignment for the PE with the received sender Id.
Good Message	7	---	- A received message at $FMon_j$ is considered good (i.e., correct) if it passes all the error checks.

4.3.2. Fault Injection

This subsection gives an overview of the fault injection capability designed according to the concept described in Section 2.2.

4.3.2.1. Mapping the OTH Fault Modes to Component Fault Modes

The smallest units of failure relevant to the design and analysis of the SPIDER protocols and services are nodes and links (i.e., computation and communication functions). In general, the functionality (or service) delivered by a component can be specified in terms of a generated sequence of output service items, each of which is characterized by a value-time tuple (i.e., a value delivered at some time) [Powell92]. From this viewpoint, the failure of a component is manifested as a corruption of the output service in the value and/or time domains.

Consider the case of one-to-one communication between a source node and a receiving node. The information arriving at the receiver is the result of the behavior of the source node and the communication link between the source and the receiver. Let $\textit{Fault Type}_{\textit{one-to-one}}$ denote the fault manifestation at the receiver, which is expressed in terms of the fault types (or "modes") of the source node and the link as follows:

$$\textit{Fault Type}_{\textit{one-to-one}} = \{\textit{Node Type}, \textit{Link Type}_{\textit{one-to-one}}\}$$

Here, *Node Type* and $\textit{Link Type}_{\textit{one-to-one}}$ are each one of good (g), detectably bad (d), or undetectably bad (u). In this detectability-based model, the classification of a component malfunction is based on the effect of the errors in an output service item as perceived by an observer using value and/or time checks to detect bad service items. A service item arriving at the receiver is good if it is coming from a correctly functioning component (i.e., that meets its service specification and has uncorrupted internal state). In a properly functioning system, a good service item always passes all the value and time checks (i.e., there are no false-positive detections). A bad service item from an incorrectly functioning component is undetectable only if it passes all the value and time checks (i.e., false-negative detection is possible). Because the error checks performed by an observer can be dependent on its mode of operation, it is possible that a particular error pattern in a service item can be interpreted differently in different modes of operation of the observer. Table 4.2 captures the relation between the source node and link fault types and the resulting fault type at the receiver. An asterisk '*' is used to indicate that a type can be any of 'g', 'd', or 'u'. Note that because of the series connection of the source node and the link, the fault type of the source node is visible at the receiver only if the link is good.

Table 4.2: Detectability-based fault manifestation at the receiver as a function of source node and link fault types

Source Node Type	**Link Type$_{\textbf{one-to-one}}$**	**Fault Type$_{\textbf{one-to-one}}$ at the Receiving Node**
g	g	g
d	g	d
u	g	u
*	d	d
*	u	u

If a node has more than one output link, the classification of the *Link Type* must be expanded to cover all possible combinations of misbehavior in the output links of the source node. For this case of one-to-many communication, we add the dimension of consistency (or "symmetry" or "agreement") of manifestations at the receivers to the definition of communication fault types. The definition of consistency depends on the context. In general, a group of observations are consistent if all the pair wise differences are within a particular bound. Consistency can be defined in terms of approximate agreement, where the differences are within a predetermined, non-zero bound. Consistency can also be defined as an exact agreement with no difference between observations (i.e., a bound of zero). A service item is consistently perceived at the receivers if it is consistent in the value and time domains. When a receiver detects an error on a service item, the error indication supersedes the actual received (or missing) service item. So, with respect to error detectability at the receivers, all detectably bad outputs are symmetric irrespective of whether the actual errors that triggered the detections were the same. For undetectably bad outputs, for which there is indication of an error, the symmetry of the errors is an important fault mode characteristic. For these, the classification is further refined to distinguish symmetric and asymmetric undetectably bad outputs. These refinements are labeled 'su' and 'au', respectively. The list below gives several examples of the *Link Type* for one-to-many communication with a refined notation based on the concepts of detectability and consistency.

g	=	all sysmmetrically good
d	=	all symmetrically detectably bad
su	=	all symmetrically undetectably bad
au	=	all asymmetrically undetectably bad
g/d	=	asymmetric with some good and others detectably bad
g/su	=	asymmetric with some good and others symmetrically undetectably bad
g/d/au	=	asymmetric with some good, some detectably bad, and others asymmetrically undetectably bad

With this scheme, the OTH fault classification in Table 2.2 can be restated as shown in Table 4.3. *Fault Type*$_{one\text{-}to\text{-}many}$ defines the behavior of the nodes and links in the OTH model in a way that is suitable for the realization of a fault injection capability.

Table 4.3: OTH fault classification based on *Node Type* and *Link Type* outputs

Fault Mode	**Description**	**Fault Type**$_{\textbf{one-to-many}}$ **perceived at the receiving nodes**
Correct	All observers receives the same correct message.	{g, g}
Omissive Symmetric	Each observer declares the message invalid, either because the message was not received or it was detectably incorrect.	{d, g}, {*, d}
Transmissive Symmetric	Each observer accepts the same incorrect message.	{*, su}, {u, g}
Strictly Omissive Asymmetric	Some observers receive the same correct message and others declare the message invalid, either because they do not receive a message or declare invalid their received message.	{g, g/d}
Single-Data Omissive Asymmetric	Some observers accept the same incorrect message and others declare the message invalid, either because they do not receive a message or declare invalid their received message.	{u, g/d}, {*, d/su}
Transmissive Asymmetric	The observers have other patterns of disagreeing observations.	All other combinations.

4.3.2.2. Design Approach

The fault injection capability must support physical and simulated fault injection. For physical fault injection, SUT faults are generated by means of environmental disturbances. In this case, the CFIMS interacts with an Environment Controller (see Figure 4.4) to coordinate the activation of the disturbances with the operation of the system, and no additional internal local action is required by the CFIMS for the generation of faults. For simulated fault injection, the Test Specification describes the desired faultload and the CFIMS has exclusive control over the generation of faults. We want a flexible and configurable simulated fault injection capability that satisfies the properties of reachability, controllability, repeatability, reproducibility, and non-intrusiveness for a faultload based on the Detectability, Consistency and Persistence (DCP) fault model (see Section 2.2) applied to fault manifestations in the value and time domains. We want to be able to inject simulated faults in any of the operational modes of a node (see Section 3.4). We also want the capability to independently control fault injection at different locations and to set up complex patterns of fault injection throughout the SUT, including scenarios of multiple simultaneous faults. The CFIMS architecture (depicted in Figure 4.4), the selection of SUT fault injection points and the design of the fault injectors are critical elements in achieving the desired attributes for the simulated fault injection capability.

The CFIMS architecture, especially its connectivity, enables a high degree of coordination between the fault injection sites at the SUT nodes. The centralized PTC and STC fault injection controllers can monitor SUT activity by means of the Function Testing process (see Figure 4.4). It is also possible to add a capability in which the controllers receive additional SUT-related event information sent out via the TCLs by the distributed Local State Monitoring or Local Fault Injector processes at the SUT nodes. With this detailed level of real-time information about the state of the SUT, the fault injection controllers can send out coordinated commands to all the local fault injectors to achieve virtually any desired fault pattern, with repeatability constrained mainly by the quality of the SUT-related information made available to the controllers.

Structurally, the SUT is composed of processing nodes and communication links, and each node is further composed of a computation module and a communication module. We would like our fault injection points to be selected and organized along the same categories of computation and communication based on the functionality targeted for disruption. Computation faults can be realized by fault injectors attached to the computation modules of the nodes, and fault injectors at the communication modules can generate communication faults. Given that the OTH model classifies faults by partitioning the fault effects space, every OTH fault category can effectively be realized by injections at the source or at the observers. Source-side fault injections, including computation and communication faults, have the advantage of being centralized at the source node, which means that local state information at the source can be used as a common reference to coordinate the activation of fault injections. A disadvantage of source-side injection is that the mapping from injected faults to a particular OTH fault class at the observers may be difficult (or even impossible) to predict because of the dependence of the classification on the state of the observers. Receiver-side fault injections can be implemented by placing fault injectors on the output signals from the receiver's communication module that are connected to inputs of the computation module. This way of injecting faults has the disadvantage of a more complicated coordination problem to control the fault injectors to achieve a particular OTH fault class, with the effectiveness determined by factors like the precision of the timing information available to the fault injection controllers. The main advantage of receiver-side injection is that, assuming that the actions of the fault injectors are adequately coordinated, there is a direct mapping from the injected faults to a particular OTH fault class. In a time-triggered system like SPIDER, it is possible to simplify the receiver-side fault injection coordination problem by using the fault injection controllers to set up the fault injectors and giving fault-activation control to the fault injectors themselves, which would then use the local state of their respective receivers (including the local time) as references to trigger the activation of faults.

The fault injectors are the only components of the fault injection system that have a direct influence on the functionality of the SUT. As such, their defining features are strong determinants of the effectiveness of the fault injection system. The purpose of a fault injector is to corrupt signals at some level in the SUT structural hierarchy. To achieve this signal corruption, a fault injector is attached to a signal line and the correct signal is replaced by a faulty signal that is different in the value and/or time domains. A fault is active when the signal on the line is not the correct one. The faulty signal can be a function of the correct signal or independent from it. While the fault is active, the characteristics of the faulty signal can be static (i.e., stuck-at) or dynamic. The duration (or "persistence") of a fault may vary from one activation to the next. The correct signal is restored when a fault becomes passive.

Because a potentially large number of fault injectors may be integrated into the SUT, they should have a simple interface and be easily configured as described in the Test Specification. The design of the fault injectors must support injection on the computation and communication modules of a node. To support the centralized and distributed patterns of injection control, a fault injector must be able to receive commands from the injection controllers and to monitor as needed the state of the node at its location. The chosen fault injector design concept must not preclude having the capability to send status messages back to the main controllers via the local TCL. The design of the fault injector must also be modular in order to support evolution of the design to allow increasing complexity of fault injection patterns.

For physical fault injection experiments, the main requirement on the fault injection capability is that the faults experienced by the SUT be caused only by the test environmental disturbances. A simple way to achieve this is to send a command to the fault injectors to disable them before an experiment. However, this solution has an associated risk of unintended activation of a fault injector during an experiment due the environmental disturbances. The only way to guarantee that the fault injectors will not be activated during a physical fault injection experiment is to remove them from the SUT.

For the first set of simulated-fault experiments intended to characterize the control system-level effects of faults in the ROBUS-2 communication system, only BIU and RMU faults are of interest and the only fault type is the fail silent (or fail stop) mode in which a node stops producing outputs when it fails [Butler08]. This failure mode provides source-guaranteed integrity (i.e., absence of improper state alteration) [Avizienis04, Paulitsch05] on the node's output service and requires simple error decision logic at the observers to validate the service: all received service items are correct and missing items are bad and detectable by a timeout check. This failure mode was selected in order to simplify the modeling and analysis of propagated fault effects on ROBUS-2 and the control system [Gray08]. In essence, the analytical models were developed assuming that there are no propagated fault effects other than those caused by the persistence of node service unavailability. To implement this kind of fault injection capability, a fault injector was attached to the reset signal at each of the BIUs and RMUs, and the activation of the injectors is directly controlled by the main fault injection controller at the PTC following the pattern of fault locations and timing specified in the Test Specification.

4.3.2.3. Current Design

The design of the Local Fault Injectors reflects the levels of abstraction as well as the fault models described in Section 2.2. The Local Fault Injectors have a general structure so they can be placed on any control or data signal within the SPIDER nodes, causing the signal to deviate from normal operation.

This architecture allows the location of the fault injectors within the nodes, also referred to as injection points, to vary. The faults have two parameters to indicate their behavior and persistence. The behavior, or fault type, specifies what values the faulty signal will undergo for the persistence of the fault specified by the activation pattern. The activation pattern and duration specify when the faulty value is used in the node operation. The fault value might take on a value independent of the original signal or might be a function of the target signal itself. The fault injection setup section of the Test Specification file (see Appendix A) defines the fault type and activation pattern of the target signals.

The software interprets the fault injection setup section of the Test Specification file. The fault injection setup data from the Test Specification file is passed from the Repository, or Data Management, to the PTC software Fault Injection Management process, where the setup data is sent to the PTC Fault Injection Controller of the hardware during the fault injection setup mode of the CCP. With the current setup, the PTC Fault Injection Controller forwards the setup messages across the PTLs to all of the PE-BIUs and also across the CCL to the STC. The STC Fault Injection Controller then forwards the setup messages across the STLs to all RMUs. Once the PE-BIUs and the RMUs receive the fault injection setup data, the Local Fault Injectors parse the data and set up the corresponding fault type and activation pattern. The fault specification capability may be expanded to allow the STC software to send setup information to the hardware when it is needed to send separate fault injection setup data to the RMUs or to eliminate the data forwarding in the STC hardware.

In the current implementation of the CFIMS, we assume the nodes are fail-silent. When errors are detected in a BIU or RMU, the outputs of the node are disabled (see Section 3.3), indicating an inherently fail-silent behavior. The fail-silent fault is implemented by asserting the reset signal for the duration of the scheduled exchange of PE messages (see Section 4.1). The start and end times of the fault must be declared in order to only affect the node for a portion of the control cycle. Within the Test Specification file two lines are used to specify the RCI (see Section 4.3.1.1) to activate and deactivate the fault. By deactivating the fault soon after the sensor and command exchange is complete, the rest of the control cycle is left for the SUT to recover for the next independent control cycle (see Section 4.1).

The architecture of the SUT can be divided into two main parts: application-level computation and communication. The computation portion of the system contains PE nodes, and the communication portion contains BIUs, RMUs, and data links. Applying failures on the nodes and links of the SUT is a plausible high-level system fault model. Observing fail-silent fault effects on the nodes provides this system-level model. After studying the effects of physical faults, it might be necessary to trace system failures to the component-level. The fault injection system is designed to be flexible for both high- and low-level injections. The depth in the lower levels of fault injection abstraction determines the quality of the results and the complexity of the analysis.

The capability to execute a round without conducting any simulated fault injection is included in the design of the fault injection subsystem. This capability allows the gathering of data to establish a base case that the fault injection experiments can be compared against. Additionally, the no-fault-injection setting is used for the physical fault injection experiments, such as in HIRF environments. The specification that the round has no fault injection replaces the fault type and activation of the fault injection setup information in the Test Specification file (see Appendix A). The PTC software sets the no-fault-injection status and the PTC Fault Injection Controller does not send any fault injection setup data, therefore, allowing the round to be completed, including the collection of the monitoring data, without injecting any simulated faults into the system.

Each SPIDER node is specified to either behave normal (0) or fail-silent (1) for each control cycle within the round (see Section 4.3). A 16-bit fault vector (or test vector) describes which nodes contain faults for a single control cycle. Similar to the fault injection setup data, the fault vectors defined in the Test Specification file are sent from the Repository to the PTC software Fault Injection Management process. Before the start of the round, the software preloads some fault vectors into a buffer in the hardware where it waits until the first control cycle of the test. Throughout the execution of the test, the PTC Fault Injection Controller reads fault vectors from the buffer to pass on to the PE-BIUs and the STC Fault Injection Controller. The STC Fault Injection Controller forwards the fault vectors to the RMUs. As fault vectors are read from the buffer, the software continues to write the fault vectors onto the buffer until all fault vectors have been sent. At the SPIDER nodes, the Local Fault Injectors receive the fault vectors and locate the corresponding bit that applies to their Node Id to either activate or deactivate faults.

4.3.3. State Monitoring

The data collected about the internal operation of the SUT is used for real-time health assessment and post-test event analyses. Figure 4.9 shows a dataflow diagram for the state monitoring function of one SUT node. Every SUT node has an embedded node monitor (ENM) that can gather state information from the local components, including a ROBUS Protocol Processor (RPP) programmed as either a BIU or an RMU, several ROBUS Links (RLs), one or more local Fault Injectors (FI), and a Processing Element (PE) if the node is a PE-BIU node. The ENM can be reconfigured to change the monitored node variables and the triggering conditions for sampling individual variables. The data samples are taken as their respective sampling conditions are triggered, and they are retained in local memory until at least one of the configured conditions for the transmission of a state message is triggered. If there are variables for which there is not a sample in memory when the transmission trigger occurs, those variables will be sampled immediately in order to complete the data sample record. The state data in memory is then packaged into a TCM and sent over the local TCL to the State Message Receiver (SMR) at the corresponding test controller (i.e., the PTC for PE-BIU nodes or the STC for RMU nodes). There, a Node Condition Monitor (NCM) examines the received state data to determine the current health status of the node. Figure 4.10 shows the state transition diagram for the current health condition assessment algorithm. The selection of health indicators (i.e., the state variables used in the node condition assessment) and the condition assessment algorithm are configurable elements of the state monitoring function. In Figure 4.10, the assessed node condition is Disabled until a new set of indicator values (i.e., a new health record) is received. When that happens, the condition is upgraded to Recovering, where it remains until there is sufficient evidence that the node is in good and stable health condition. The definition of what constitutes a good and stable health condition is part of the system configuration. As currently configured, good and stable health is indicated by a continuous sequence of health records with content matching a predefined set of good-health indicator values for a pre-set minimum time duration. When the transition condition for the Recovering state is satisfied, the condition is upgraded to Restored. The node condition is set back to Recovering if there is evidence that the node is not in good health. As currently defined, this is triggered by a timeout in the gap between health record updates or a health record content that does not match the good-health indicator values. The assessed node condition is forwarded to the SPIDER Health Monitor at the local test controller and also merged with the received state data to form an output node state record to be collected by the software.

More detailed information about the state monitoring function is given with the design descriptions for the SUT and CFIMS nodes in sections 9 through 12, and also in Appendix B, which provides a comprehensive list of the current set of state variables in the node state records.

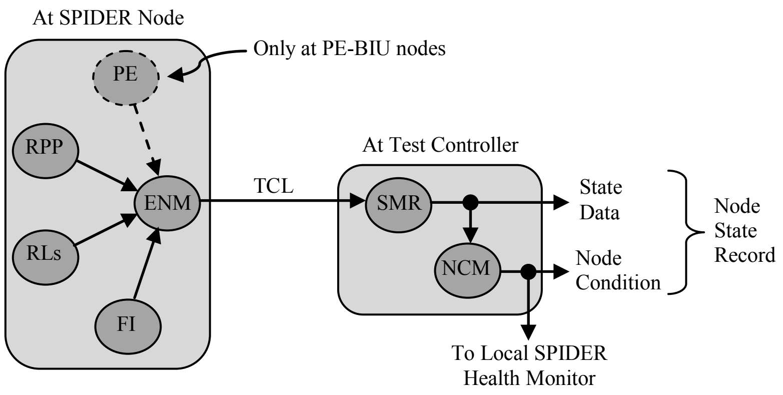


Figure 4.9: Dataflow graph for a node state monitoring lane

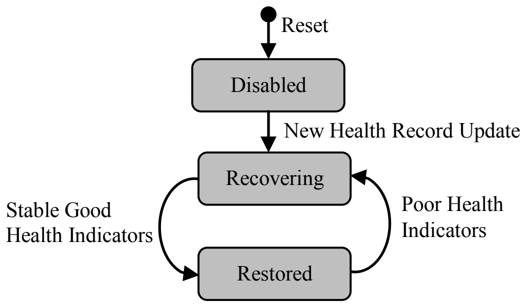


Figure 4.10: State transition diagram for node condition assessment

4.3.4. Round Control

The purpose of the CFIMS Round Control function is to provide common direction for global coordinated interaction among all the SUT and CFIMS processes. The main Round Control processes in the current system are implemented in hardware at the PTC and STC, as shown in Figure 4.4. The Controller Coordination Protocol (CCP) executed over the CCL allows the distributed round control processes to present a tightly coordinated, SUT-status-aware control interface to the other hardware and software processes. Figure 4.11 shows the major mode transition graph for the Round Control function. After a reset, the system remains idle until the software commands the start of a round. The first step after a software enable is for the hardware round control processes to synchronize their actions. When that has been accomplished, the controllers command SPIDER to initialize and get ready to run when commanded, which currently involves enabling ROBUS to initialize and enter its Clique Preservation mode and for the PEs to monitor the PTLs for incoming round setup commands. The CFIMS state monitoring function begins tracking the state of the SUT at this time. The fault injection (FI) processes, including the fault injectors at the SUT nodes, are then configured according to the Test Specification. Next, the SUT function, realized mainly at the PEs, and the function testing process at the PTC are also configured. The SUT function is enabled when the system setup phase is completed. The system executes until a workload or fault injection completion condition in the Test Specification is satisfied, or until some error condition is detected, including the possibility of a permanent SPIDER failure. After the CFIMS processes have safely ended their operations, the system returns to the idle state until the control software is ready for another round.

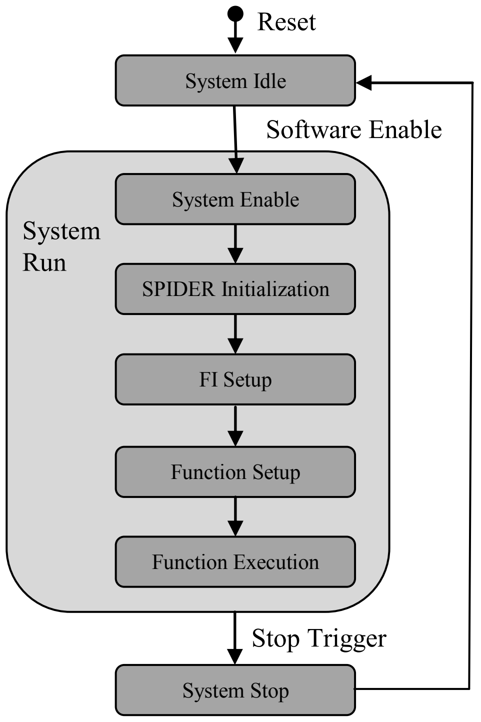


Figure 4.11: Major mode transition diagram for CFIMS

The Round Control function is supported by a SPIDER Health Monitoring function whose main purpose is to determine when the SUT has been successfully initialized and when it has experienced an unrecoverable failure. The health of the SUT is monitored at the PTC using data from the PE-BIU state monitors and the Function Testing process, and at the STC using RMU state monitor data. As currently configured, there are three health monitoring phases. The first health monitoring phase is during the SPIDER Initialization round control mode, where it must be determined when the SUT has been successfully initialized. In the current implementation, this is indicated by the node condition reported by the state monitors at the PTC and STC reaching the Restored state for every active SUT. The second health monitoring phase begins when SPIDER is initialized and extends until the SUT begins executing its application. During this time, which corresponds to the interval of the FI Setup and Function Setup round control modes, the node condition reported by the state monitors at the PTC and STC for every active PE-BIU and RMU node should remain in the Restored state (see Figure 4.10) for the complete time interval because no faults are being injected into the SUT. A SPIDER failure is immediately declared if the node condition for any active node is other than Restored at any time during this phase. The third health monitoring phase extended for the duration of the Function Execution mode of the Round Control function. During this mode, SPIDER is considered to be working properly as long as an attempted re-initialization does not last longer than a theoretically derived maximum based on system fault assumptions and a timing model of the recovery dynamics. For the current implementation, the mode of the FTmr is used as an indirect means to determine when the SUT is re-initializing. A SPIDER failure is declared if the FTmr remains in the Initialization mode (see Figure 4.7) longer than expected under the conditions covered by the fault assumptions.

In addition to a common mechanism for distributed hardware and software process coordination, the CFIMS Round Control function also provides a global time-reference service used to time tag observations for use in post-test event analyses. This service leverages the CCP messages between the

PTC and STC controllers during the system setup phase (i.e., from major mode System Enable to mode Function Setup), as well as the synchronization messages from the FTmr during the Function Execution mode, to synchronize a pair of Round Timers (RTmrs) at the PTC and STC. The Round Time has two elements: the Interval Count (IC) and the Interval Time (IT). The Interval Count is the number of times the RTmrs have been synchronized by messages sent across the CCL since the beginning of the round. The Interval Time measures the time elapse since the last synchronization event. The synchronization dynamics of the RTmrs is closely tied to the CCP. Section 5 of this report provides a detailed description of the CCP using event sequence diagrams.

4.3.5. Data Collection

As stated in a previous section, the purpose of an experiment is to gather observations about the response of the SUT when operating under conditions determined by the configuration, workload and faultload specifications. From Figure 4.4, it is easy to see that the CFIMS can be viewed abstractly as providing a user service that reads a Test Specification and outputs a number of files containing the corresponding experimental data. In its current version, the main outputs to the system user include data generated by the Round Timer (RTmrs), Function Timer (FTmr), Function Monitors (FMons) and State Monitors (SMons). The data from each output stream is organized into records (or snapshots) about the activity happening at the source process at a particular point in time. Each output record is time tagged with the RTmr value at the time the record is generated. The data records from the RTmrs themselves (one at each controller) capture the values of IT and IC every time the RTmrs are synchronized. In addition to the raw experimental data, the system also generates a test log with round control-related entries, including the condition that triggered the round stop. Test control software at the PTC and STC saves all the output data to files in the Repository. Appendix B gives a detailed description of the output files.

4.3.6. System Configuration

From a configuration perspective, the system consists of the SUT, PTC, STC, Repository and Environment Controller. All of these have hardware and software elements that can be changed to meet experiment requirements. The interaction between the repository and the test controllers is determined by data transfer protocols implemented in software. Likewise, a software-implemented protocol is used to coordinate the operation of the Environment Controller with the activities at the SUT and the CFIMS. These software protocols, which are not expected to change significantly throughout the duration of the research project, are described in Section 13 of this report. The execution platform for the SUT and the test controllers will eventually transition from RSPP1 to RSPP2 for the reasons described earlier in this report (see Section 4.2). It is expected that the transition may have an impact on the software and hardware implementation of the SUT and the test controllers, but the functionality and configurability of the system should be largely unaffected. Also, as described in Section 2.4, the system configuration strategy involves the ability to set up the system as desired at any level in the development, including design, synthesis, and runtime. The need to modify the system design is determined by whether or not the requirements of an experiment can be met with the existing system configurability for synthesis and runtime. The selected system architecture is such that most design changes should involve functionally related processes (e.g., fault injection processes in hardware and software) and the system capabilities should be expandable mostly through localized design changes without significantly increasing the overall design complexity. This enables the introduction of a high degree of synthesis and runtime configurability for a particular design, which extends the longevity of the design and reduces the frequency and extent of design changes.

Accordingly, the focus here is on the synthesis and runtime configurability for the initial system design of the SUT, PTC and STC. In that context, configuring the system consists of assigning values to structural and behavioral design-implementation parameters for architectural components like ROBUS and the PEs at the SUT, and at the CFIMS processes of SUT function testing, fault injection, SUT monitoring, round control and data collection. The current approach to determine appropriate parameter values is to use system architecture and implementation models that capture desired high-level requirements (e.g., number of PE-BIUs and RMUs, desired ROBUS cycle rate, number of ROBUS cycles per control cycle), as well as lower-level implementation requirements in the form of explicit parameter constraints (e.g., valid parameter value ranges) and requirements (e.g., ROBUS message width), execution platform specifications (e.g., clock rates, oscillator drift rates, point-to-point communication delays and timing uncertainty), functional design constraints (e.g., minimum input-output delay for certain components, minimum reset delays), and environmental constraints (e.g., maximum fault duration). Determining the values for the synthesis parameters is a two step process. In the first step, a model solution is found that meets the requirements and constraints and is optimized for performance where possible. In the second step, the parameter values are computed so the synthesized system matches the model solution. For the most part, the synthesis-time hardware parameters correspond to structural and behavioral implementation details like register and counter bit-widths, data buffer sizes, time delays, timeout durations, and event count triggers. The compilation-time software parameters set up how the software allocates the available memory and how it interacts with the CFIMS hardware processes, the Repository, the Environment Controller, and the operator interfaces (i.e., display and keyboard).

The Test Specification gives the runtime parameters, which cover two aspects of the system: SUT function and fault injection. The runtime parameters for the SUT function include the selection of which PE-BIUs and RMUs will be active during the round, the function assignment (IO or Processing) for each PE, the number of ROBUS cycles per control cycle, the PE communication schedule over the duration of a control cycle and the application level data to be exchanged by the PEs. For a simulated fault injection round, the runtime parameters set up the fault injectors by selecting the fault type and activation pattern for each injector. Depending on the type of fault to be used, the Test Specification may also include the necessary information to allow the fault injection process at the PTC (i.e., the Primary Fault Injection Controller, PFIC) to centrally and directly control the injection of all the faults. Appendix A provides a detailed description of the Test Specification.

Setting up the parameters requires a complete system configuration analysis to ensure that the values are compatible with one another and that the experiment requirements are satisfied. This is currently performed using a combination of automated and manual analyses covering the SUT and CFIMS. The automated analysis tool is based on the ROBUS-2 configuration analysis program [Torres05B] with expanded capabilities to cover most of the PE and the CFIMS synthesis-time parameterization for hardware processes. Manual analysis is used for the remainder of the hardware synthesis parameters and for all the software and runtime parameters. Due to the large number of system configuration parameters, there is significant interest in completely automating the process of analysis and computation of parameter values in order to reduce the likelihood of configuration errors and to simplify the reconfiguration of the system to adapt to evolving research needs.

5. Hardware Controller Coordination Protocol

The hardware-implemented CCP enables the PTC and STC Round Control processes (see Section 4.3) to coordinate their actions and provide overall round-level coordination for other hardware and software processes at the PTC and STC. The CCP is implemented in hardware to ensure fast response to local events. A hardware implementation also enables the generation of events with very high synchronization precision (equivalently, very low timing skew) at the PTC and STC. With these events, the Round Controllers can drive a pair of synchronized Round Timers (RTmrs) that constitute a common distributed time reference and are used to time tag output observations.

Figure 5.1 shows the round execution flow at the major-mode level. After a reset, the system remains idle until the PTC and STC software enable the execution of a round. In the System Enable mode, the hardware round controllers confirm that they are both active, establish initial synchronization for coordinated action, and reset the PTC and STC hardware processes to prepare for the round. Next, the round controllers issue a command to reset and initialize SPIDER. After SUT initialization is confirmed, the controllers command setting up of the fault injection system and the SUT function, including the SUT testing processes at the PTC. The SUT is then enabled to begin full execution. Once enabled to run by the software, a round will continue until the specified fault injection campaign is complete, the execution of the SUT reaches a predefined endpoint, SPIDER experiences an unrecoverable failure, or some other error condition is detected. The following subsections describe the CCP activities for each of the major modes.

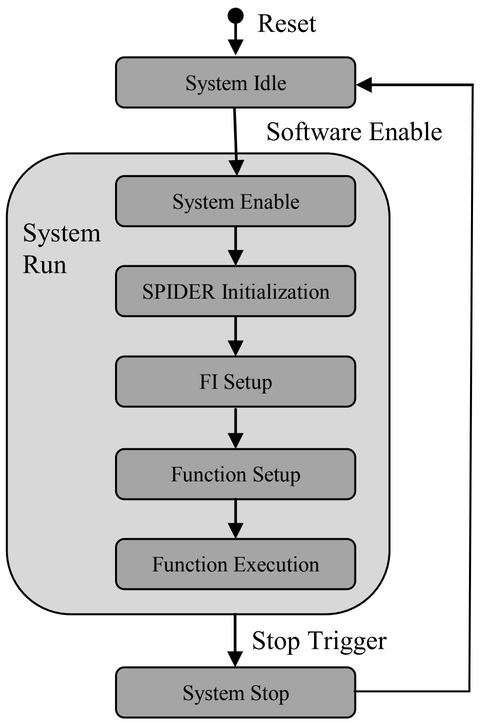


Figure 5.1: Major mode transition diagram for the CFIMS

5.1. Normal Run Modes

The CCP is described using event sequence diagrams as shown in Figure 5.2. Dots on a vertical line represent events at the corresponding hardware or software process. Horizontal and slanted arrows connecting dots represent the propagation or communication of events between processes. Time flows down along the vertical lines in the figure. The process labels at the top are for the following processes.

- RCtlr = Hardware Round Control Process
- RTmr = Round Timer
- HMon = SPIDER Health Monitor
- FTmr = Function Timer
- FMon = Function Monitor
- FIC = Fault Injection Controller
- SMon = State Monitor
- SW = Software

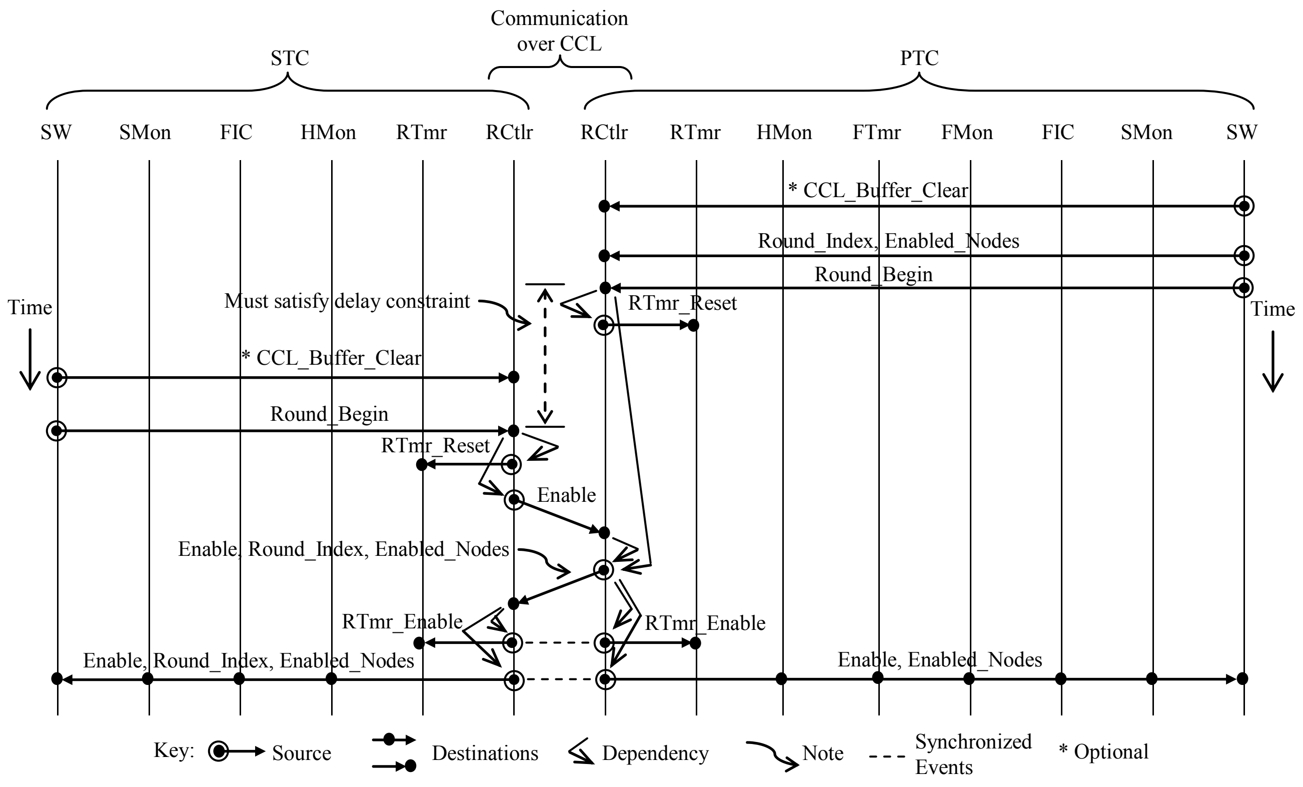


Figure 5.2: Event sequence for normal System Enable mode

Figure 5.2 illustrates the normal sequence of CCP events (i.e., with no unexpected round-stop triggers) for the System Enable mode. The system enters this mode from the System Idle mode. The optional CCL_Buffer_Clear signal events sent from the software to the round controllers are used to clear the round controllers' input buffers for CCL messages. This action may be necessary to delete any unprocessed messages resulting from an abnormal round stop scenario, as described in the next subsection. To start the PTC, the software loads the Round_Index and the Enabled_Nodes specifications for the round. The Round_Index is generated only at the PTC and is used to uniquely identify the data sent to the Repository at the end of the round. The Enabled_Nodes specification indicates which PE-BIUs and RMUs will be active during the round. The Round_Begin signal from the SW enables the RCtlrs, whose first action is to reset the RTmrs to prepare for the round. The PTC RCtlr then monitors the CCL for the arrival of the Enable message from the STC which is expected to arrive before the PTC RCtlr times out and triggers a stop condition. Upon reception of the Enable message, the PTC RCtlr replies with an Enable message carrying the Round_Index and Enabled_Nodes values. Using the known communication delay over the CCL, the PTC and STC can use the times of transmission and arrival, respectively, to synchronize the RTmr_Enable signals to start the RTmrs (see Appendix B in [Torres05A] for a detailed description of how this is done). The RCtlrs also enable all of their respective hardware and software processes.

The normal event sequence for the SPIDER Initialization mode is shown in Figure 5.3. To achieve the initialization, the CFIMS takes advantage of SPIDER's built-in fault recovery capability by using the local fault injectors to trigger the re-initialization procedure. To do this, the PTC RCtlr issues a command to reset all the fault injectors embedded in the SPIDER nodes, as a result of which the fault injectors also reset all the PE-BIU and RMU nodes. The reset is cleared only for the nodes specified in the Enabled_Nodes variable, and the others remain disabled for the duration of the round. As the enabled nodes begin operation, the Embedded Node Monitors (ENMs) start sending snapshots of the state to the SMons at the PTC and STC. When the node condition monitors at the SMons indicate that the enabled nodes are in the Restored state (see Section 4.3.3), the HMons signal the RCtlrs that SPIDER has completed the initialization and is ready for operation. As a result, the RCtlrs exchange Ready messages, resynchronize the RTmrs and signal the software that SPIDER is initialized and ready.

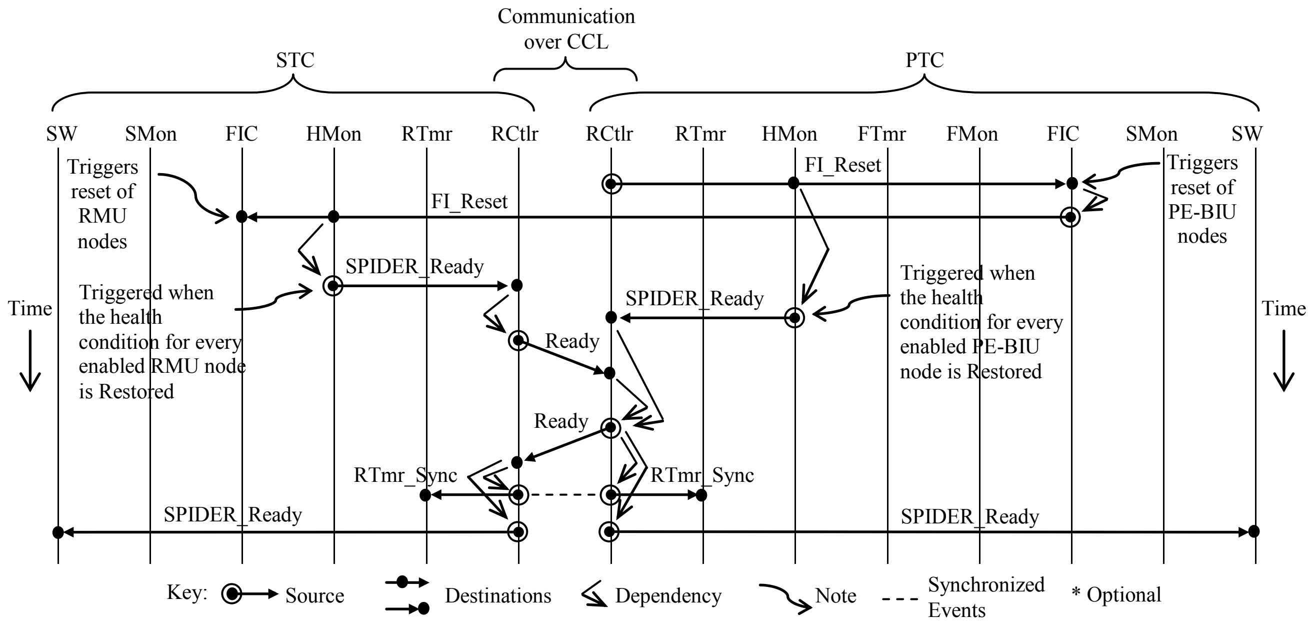


Figure 5.3: Event sequence for normal SPIDER Initialization mode

Once SPIDER has been initialized, the next step is to configure the fault injection and SUT functions for the round. Figure 5.4 shows the normal event sequence for the Fault Injection Setup mode. The procedure is started by commands from the RCtlrs to the FICs, which handle all the interaction with the software to load the fault injection (FI) configuration information given in the Test Specification file. In the current implementation, all the FI setup information is loaded at the PTC and forwarded to the STC as needed. Although currently there is no setup interaction between the SW and FIC at the STC, that capability is available as a future design option. When the STC FIC has finished its local setup procedure, the STC RCtlr sends a message to the PTC indicating that it is ready and waiting for the PTC to finish setting up the fault injection and SUT function testing. The STC Start message remains buffered at the PTC RCtlr until the setup procedure is complete.

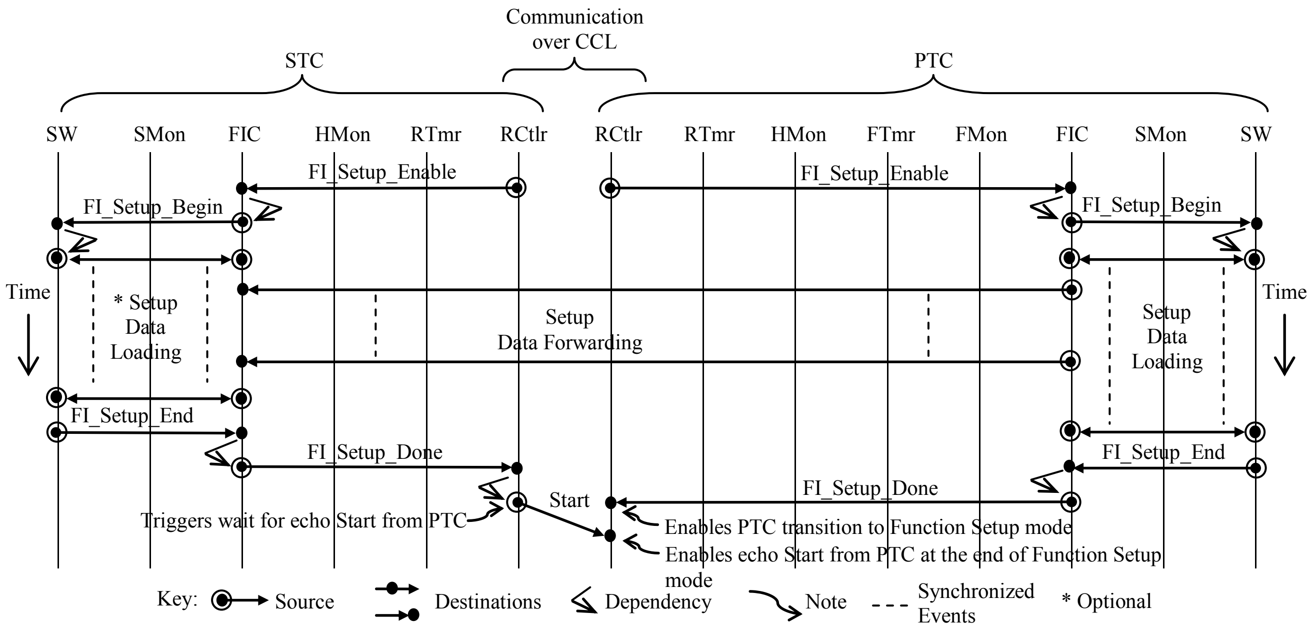


Figure 5.4: Event sequence for normal Fault Injection Setup mode

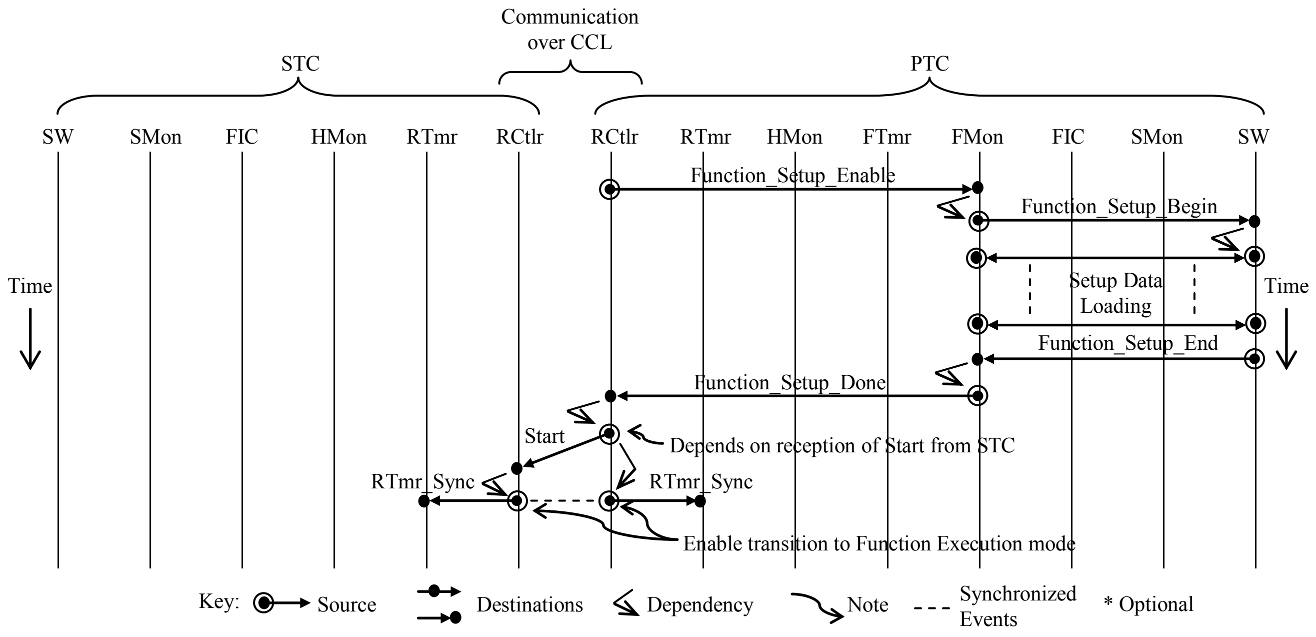


Figure 5.5: Event sequence for normal Function Setup mode

As shown in Figure 5.5, the function setup procedure consists of a data-transfer interaction between the PTC SW and the FMons to load the setup given in the Test Specification. Note that the PTC is directly connected to the PE-BIU nodes (see Figure 4.4 in section 4.3). The STC does not receive any SUT function setup data as the PEs are the only SPIDER elements involved. When the FMons are finished setting up for the round, the PTC sends a Start message to the STC and the RTmrs are resynchronized.

Figure 5.6 shows the event sequence for the Function Execution mode. In this mode the PEs execute the application and interact with the FTmr and the FMons while the FICs inject faults and the SMons collect internal SUT state data. After the RCtlrs enable the execution of the SUT application function, the PEs synchronize to ROBUS and to each other through the FTmr. When the FTmr achieves initial synchronization to the PEs, it generates a Sync event that is distributed throughout the system as a common timing reference. During the Function Execution mode, the RTmrs are synchronized with respect to the FTmr Sync events. Depending on the particular system setup, the software may load SUT function testing and fault injection data during this mode. In addition to that, the SW receives execution data records from the RTmrs, FMons and SMons to be used for post-test analyses. Normally, this mode continues until either the SUT function reaches a predetermined endpoint or the FI injection subsystem completes the injection campaign specified for the round.

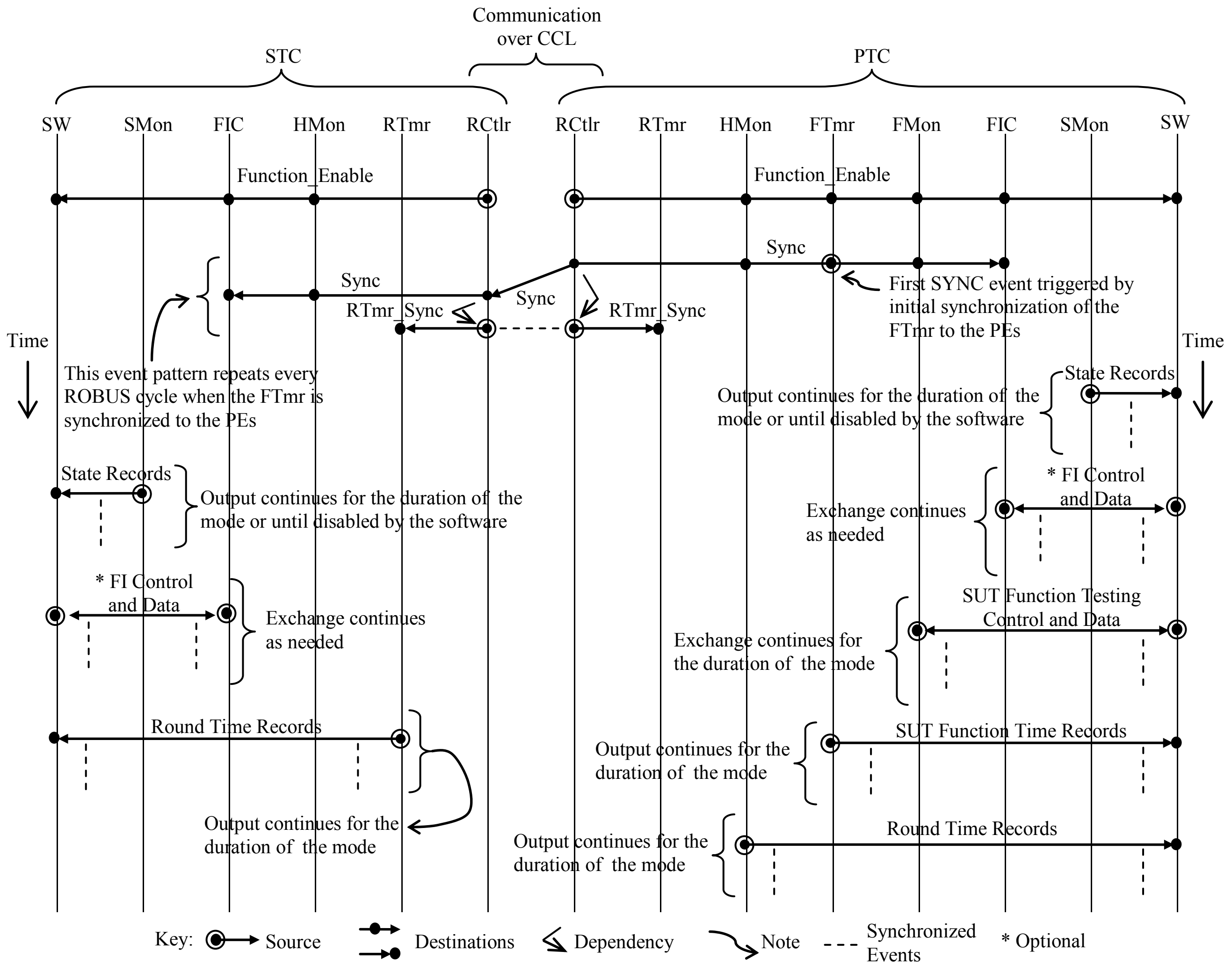

Communication over CCL
STC
PTC
SW
SMon
FIC
HMon
RTmr
RCtlr
RCtlr
RTmr
HMon
FTmr
FMon
FIC
SMon
SW
Function_Enable
Function_Enable
Sync
Sync
Sync
RTmr_Sync
RTmr_Sync
First SYNC event triggered by initial synchronization of the FTmr to the PEs
Time
Time
This event pattern repeats every ROBUS cycle when the FTmr is synchronized to the PEs
State Records
Output continues for the duration of the mode or until disabled by the software
State Records
Output continues for the duration of the mode or until disabled by the software
* FI Control and Data
Exchange continues as needed
* FI Control and Data
Exchange continues as needed
SUT Function Testing Control and Data
Exchange continues for the duration of the mode
Round Time Records
Output continues for the duration of the mode
SUT Function Time Records
Output continues for the duration of the mode
Round Time Records
Output continues for the duration of the mode
Key:
Source
Destinations
Dependency
Note
Synchronized Events
* Optional

Figure 5.6: Event sequence for normal Function Execution mode

5.2. Stop Scenarios

The goal of the System Stop mode is to halt system operation as soon as possible after normal completion of execution or the occurrence of an unexpected error condition, and then to guide the system to a graceful and coordinated stop with whole data records properly saved and the system returned to a safe state from which it is ready to begin another round. Figure 5.7 shows the event sequence for a normal System Stop mode in which the stop trigger is reaching a predefined endpoint for fault injection or SUT function execution. Both of these events are normally triggered at the PTC, where the response of the RCtlr is to command an execution stop of all local hardware and software processes and to inform the STC of the stop condition. In this scenario, the PTC is referred to as the stop initiator and the STC is the stop follower. When the STC RCtlr receives the Stop message, it issues a command to stop all local processes and acknowledges reception of the Stop message by echoing it back to the PTC. When the PTC RCtlr has confirmation that the STC and all the local processes that directly interact with the SUT have stopped, it disables the RTmr and informs the SW that all hardware processes have stopped. The STC RCtlr performs a similar action. The SW processes continue execution until they have collected all the data records.

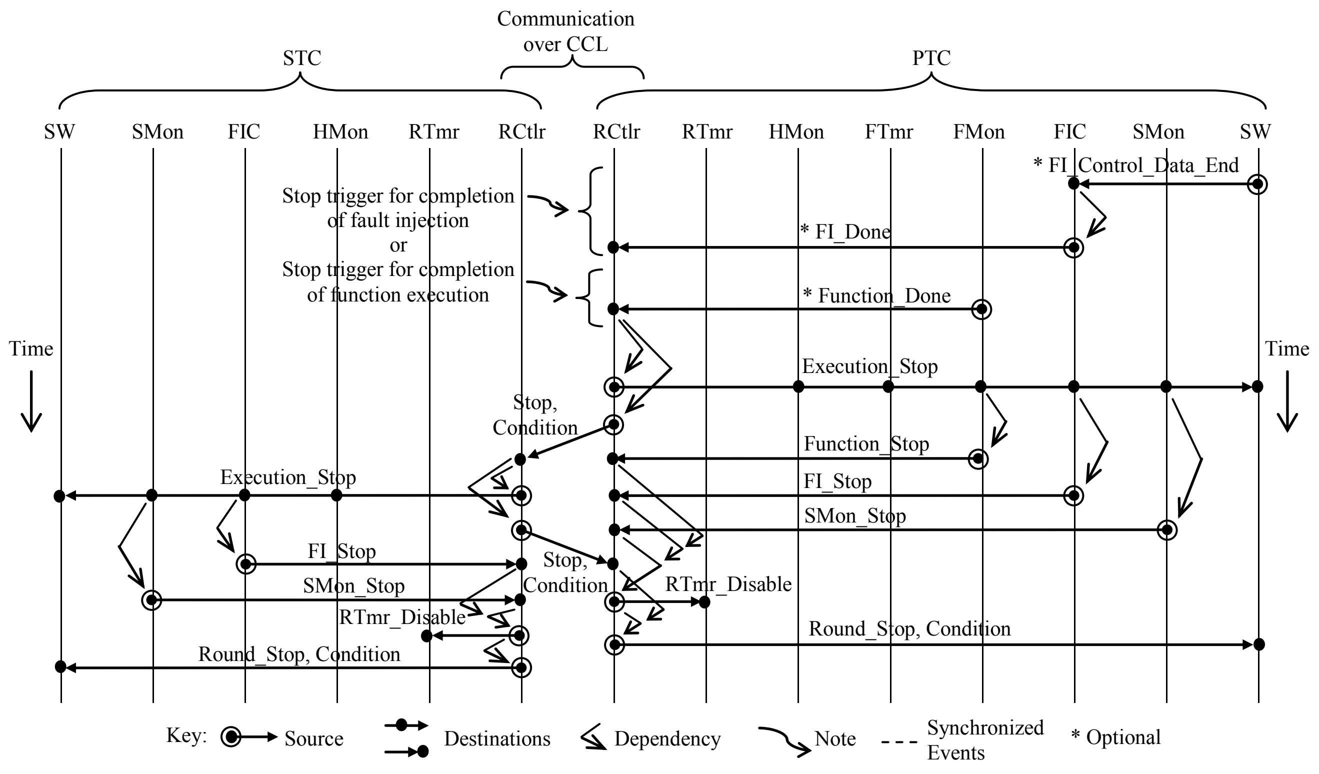


Figure 5.7: Event sequence for System Stop on completion of PTC fault injection or function execution

Figure 5.8 shows a System Stop event sequence for a trigger originated at the STC. In this scenario, the STC takes the role of stop initiator and the PTC is the follower.

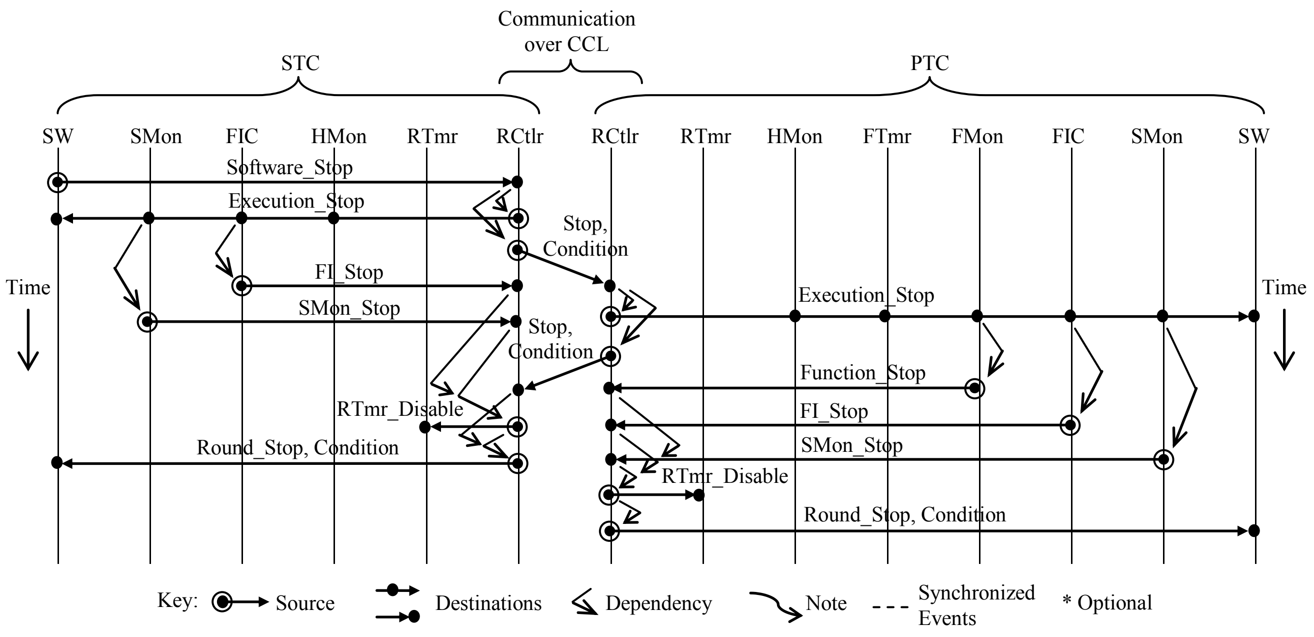


Figure 5.8: Event sequence for System Stop on STC software stop

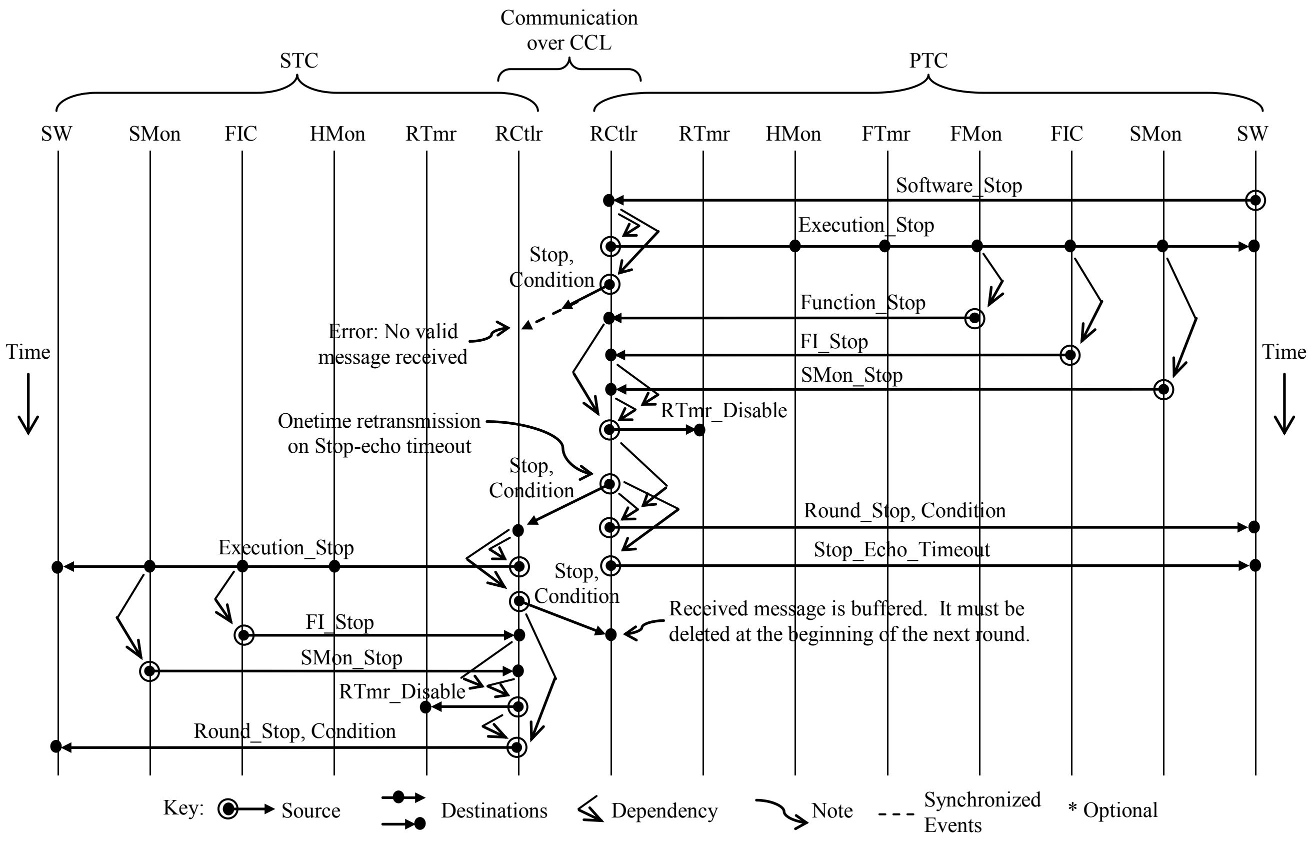


Figure 5.9: Event sequence for System Stop on PTC software stop and with initiator Stop message error

The event sequence in Figure 5.9 is for a case of a stop trigger at the PTC and a CCL communication error for the first Stop message. After stopping the local hardware processes and sending a Stop message to the STC, the PTC RCtlr times out waiting for the echo from the STC. This timeout can be due to persistent or transient CCL faults in one or both communication direction. A persistent CCL fault can only be handled by the system software or the operator. In case of a transient fault in the PTC-to-STC CCL direction, a message retransmission is likely to succeed. A transient fault in the STC-to-PTC CCL direction is not critical to properly stop the round at the PTC and STC. The specified PTC RCtrl response to a Stop echo timeout is to resend the Stop message once and then inform the SW of the timeout condition so that appropriate action can be taken if the STC does not receive the second message. For the scenario in Figure 5.9, the second Stop message is received by the STC, which then follows the normal sequence of events to stop its execution. Because the PTC is in an idle state when the STC Stop message arrives, the message remains buffered until read or deleted. To prevent a false round start, this message must be deleted as described previously in regards to Figure 5.2.

6. Data Links

This section describes the design of the custom physical data links (DLs) for point-to-point communication between hardware processes in the SUT and the CFIMS. These custom DLs are used in the ROBUS Links (RLs) interconnecting the ROBUS RPPs, and the Test Control Links (TCLs), which include the Primary Test Links (PTLs) between the PTC and the PE-BIU nodes, the Secondary Test Links (STLs) between the STC and the RMU nodes, and the Controller Coordination Link (CCL) between the PTC and the STC. High-level overviews of the SUT ROBUS and the CFIMS are given in sections 3 and 4 of this report. Figure 4.4 illustrates the process architecture of the SUT and the CFIMS, and identifies the data links for hardware process communication. The software processes running on the CPUs of the Reconfigurable SPIDER Prototyping Platform (RSPP) (see Section 4.2) use standard solutions to communicate with other entities (e.g., Ethernet and RS-232 for communication with software processes running on other nodes, and the ISA and/or PCI bus for communication with hardware processes running on local FPGAs).

The DLs can be viewed as low-level functions that provide point-to-point data transfer services between hardware processes running on RSPP FPGAs. The DLs were developed to be generic and reusable designs that satisfy the requirements of the SUT and CFIMS processes within the constraints imposed by the available RSPP resources. The RLs are used exclusively by the RPPs to communicate with one another using ROBUS Messages, which have a predetermined format and fixed message length (see Section 3). To support the various RPP communication modes, the RLs require deterministic access latency at the transmitter end and RPP-to-RPP communication delay with known upper and lower bounds. There is no inherent performance requirement for any particular level of throughput from the RLs as the RPPs can be configured to support practically any message rate, but it is desirable to have links that will not be significant performance bottlenecks for any likely SUT configuration. It is also desirable for the RLs to have high error-detection coverage and very low intrinsic bit-error-rate (BER). High error-detection coverage is intended to minimize the likelihood of transmissive faults (i.e., undetected faults; see Section 2.2) due to the communication links. Low communication BER prevents frequent nuisance faults that may activate the SUT fault tolerance mechanisms even in benign environments. Low intrinsic BER also supports the presumption that observed communication errors during an experiment are due to injected faults and not due to poorly designed links.

For the CFIMS TCLs, a DL must have the capability of being shared among multiple source hardware processes and also be able to handle messages of different lengths. From the overview in Section 4 (see Figure 4.4), some nodes have multiple processes that communicate with one or more processes at other nodes. Given the limited RSPP communication resources, including a small number of communication ports, the preferred approach to realize all the needed functional communication is to provide mechanisms that enable multiple source processes to share DLs, as well as providing mechanisms that allow receiving processes to accept only specific messages. In addition, because different processes have different data communication needs and the size of the data transfers initiated by a source process may vary throughout an execution round, it is desirable to have DLs with the capability to handle messages of varying lengths as individual and independent atomic transactions with whole-message integrity guarantees.

The design of the DLs must take into consideration the available RSPP resources that can be used in solutions to the node-to-node data transfer problem. The communication channels of a node are intended for serial communications. Each channel has a transmitter and a receiver for physical digital signaling between nodes. At the FPGAs, which are the endpoints of the communication links and the execution hosts for the custom hardware processes, there are a number of clock signal generators with frequency multiply and divide capabilities, but there are no built-in capabilities to extract a reference sampling clock from a self-clocked serial communication signal (i.e., there are no suitable digital phase-lock loops). However, the FPGAs can implement synchronous logic circuits running at very high clock rates, and the ability of the FPGA logic circuitry to quickly recover from metastable states results in an extremely low probability of processing errors caused by asynchronous sampling of input signals [XAPP077].

We are interested in DLs that offer high availability (i.e., readiness for correct service) and high integrity (i.e., absence of improper system state alterations) [Paulitsch05]. This can be achieved with designs that have high reliability (i.e., low probability of communication errors), high error-detection coverage (i.e., low probability that an error will remain undetected), and fast error recovery, which requires low error-detection latency. In developing the DLs, it is assumed that there are no faults at the transmitter or the receiver, and the focus is on handling message corruption in the communication path between the two. The principal sources of errors in the context of a DL are signal distortions in the point-to-point communication path from the serial transmitter to the receiver, and signal sampling errors at the receiver. Given that the RSPP FPGA clock signals have low drift rates (specified as ±100 parts per million, ppm, for RSPP1) and low jitter (estimated at just a few picoseconds), it is expected that the transmitter and receiver circuits will have a relatively small impact on the overall communication error rate. The largest error contributor is expected to be signal distortions in the communication path due to, for example, intersymbol interference and noise. The chosen approach for minimizing signal distortions is to operate at a communication signal rate (i.e., the baud rate) much lower than the channel signal bandwidth, and to provide shielding for electromagnetic interference (EMI) noise (for example, by using optical fibers or twisted-pair cables with optional metallic shields for increased noise immunity). Achieving high coverage and low latency for error detection is handled by the chosen signaling technique, message formatting and design of the transmitter and receiver.

The chosen data link designs for the RLs and TCLs are based on the work presented in [Torres08B], where the fundamental concept for channel signaling is to apply Manchester coding to a non-return-to-zero (NRZ) (i.e., unencoded data) bit stream. Figure 6.1 illustrates the encoding of NRZ bits. Every Manchester bit consists of two half-bits, with the first half matching the level of the NRZ bit and the second half having the opposite level. A characteristic of a Manchester encoded bit stream is the presence of a mid-point transition in every valid bit. Transitions at bit boundaries are present only when successive bits have the same value. As shown in Figure 6.1, Manchester encoding is a biphase coding technique in which the phase of the signal relative to a reference clock indicates the corresponding value of the data bit. From this perspective, phases of 0^0 and 180^0 correspond to bit values of 1 and 0, respectively. This maximization of nominal-phase separation between code symbols serves to reduce the susceptibility to signal distortions and, thus, enhance the overall communication BER. The benefits of the Manchester code include self-clocking (in the sense that the clock signal is embedded with the data), no DC frequency component (which enables communication without direct physical attachment of DL transmission and reception components, i.e., the DLs can serve to contain the propagation of faults and thus define the boundaries of fault containment regions, FCRs), and simple bit-level error detection (as a comparison of the signal levels before and after the mid-point of a bit can detect all bit errors that do not invert both levels simultaneously) [Stallings94]. One significant disadvantage of Manchester encoding is that it requires twice as much signal bandwidth as NRZ for the same data rate.

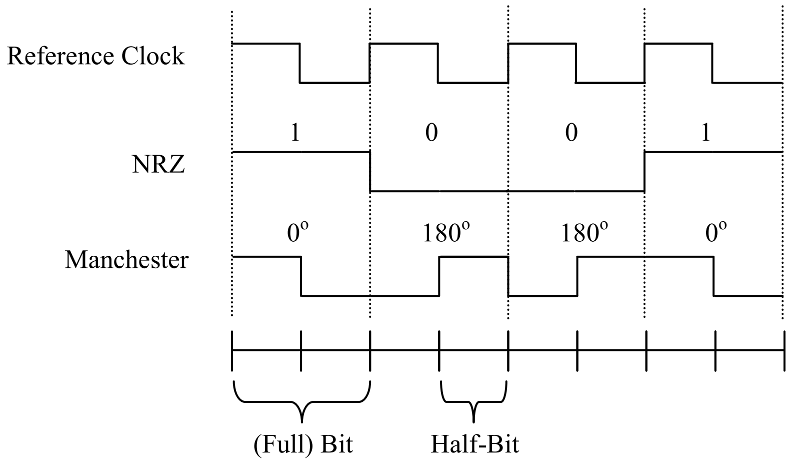


Figure 6.1: Manchester encoding of NRZ data bits

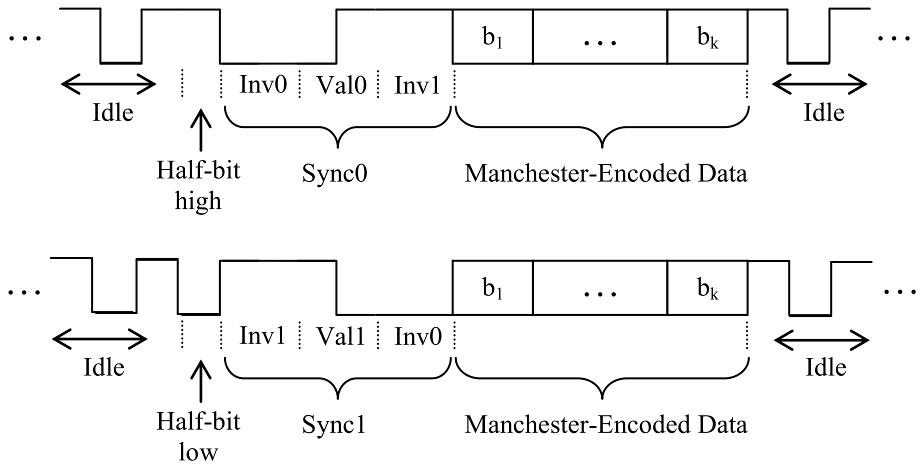


Figure 6.2: Signaling structure for the transmission of a data word (with Sync0 and Sync1)

Figure 6.2 illustrates the signaling structure for the communication of a k-bit data word. At the lowest physical signaling level, the objective is to transfer a bit-string payload from the transmitter to the receiver. Any content-specific formatting of the payload word is irrelevant to this objective. The payload data word is serially transmitted one bit at a time in some predetermined sequence. The payload word is preceded by a synchronization (or "sync") pattern whose purpose is to identify the beginning of a payload word and to provide a timing reference to sample the Manchester encoded data bits. A sync pattern can be viewed as a special code symbol different than the Manchester code symbols, but sharing the characteristic of a mid-point level transition. As shown in Figure 6.2, a sync pattern consists of an invalid-valid-invalid code sequence where the first and third bits have opposite NRZ-encoded values (and so, in a strict sense, are both invalid relative to the Manchester code), and the level transition in the valid second bit provides the timing reference for data sampling. The polarity of the half-bit immediately preceding a sync pattern is forced to be such that there is a level transition at the beginning of the sync pattern in order to minimize the uncertainty about the starting point of a sync pattern and to increase the message integrity by rejecting any sync pattern that does not meet a strict timing criteria. Both of the two possible sync patterns (i.e., Sync0 and Sync1) serve equally well for the purposes of payload demarcation and timing reference, and this flexibility in the selection of a sync pattern provides an opportunity to efficiently send a control bit with each data transmission (for example, as a tag to mark the beginning of a word sequence or to identify the word content as command or data). At this signaling level, the sequence of a sync pattern followed by a data word is called a word frame. During the idle interval between word frames, a DL sends a constant-phase clock signal corresponding to a stream of Manchester-encoded 1's or 0's. The inter-message (or "inter-word-frame") gap, which is delimited by the end of one word frame and the sync pattern of the next, must last at least one full bit-time to accommodate the forced half-bit before a sync pattern.

Signal analyses performed at the time of the work reported in [Torres08B] estimated the bit-error probability at less than 10^{-9}, or less than one error per billion bits, for this signaling scheme using the RSPP resources under worst-case conditions. Given that it is highly unlikely that the system will operate under worst-case conditions except, possibly, during fault injection experiments, the communication BER was assessed to be better than adequate for the intended use of the system.

After low BER, the next critical requirement for the design of the DLs is high error-detection coverage. As it is expected that no single technique will provide sufficient error coverage, the DLs are designed with a multi-layer approach in which different error checks are applied at each level in the design hierarchy. Some of the basic error checks are introduced here. Other design-specific checks are presented in later subsections describing the DL designs.

The first layer of error protection is to monitor the signal waveform at the receiving end of a DL. A normal incoming signal has three major states (sync, data, and idle) with only one valid major-state transition sequence (idle $\rightarrow$ sync $\rightarrow$ data $\rightarrow$ idle). If we define the level and duration of a signal between level transitions as a (waveform) symbol, then, all together, there is only a small set of strictly-defined valid symbols and a few valid symbol sequences. Any deviation from these is an indication of a transmission fault. Waveform monitoring is described further in the next subsections.

As previously described, the Manchester code offers an opportunity for a simple and effective error check based on the comparison of signal levels before and after the mid-point of an encoded data bit. An error is detected whenever the levels are the same. This error check can be applied to each coded bit in the data section of a word frame, and it can detect all bit errors that do not invert both levels simultaneously.

The other general error detection technique used in the DLs is to add redundancy to the data in the form of a cyclic-redundancy-code (CRC) bit sequence (also referred to as a frame check sequence, FCS) to enable detection of data corruption during transmission [Ramabadran88, Koopman04]. CRC checks are applied to individual word frames and also to blocks of data words transmitted over multiple word frames.

Given a DL design with high reliability, the availability can be improved by incorporating techniques that offer short error-detection latency and fast recovery. One option for enhanced end-to-end DL error detection is the use of redundancy in the designs of the DL transmitter and receiver. Transmitter-side error detection can be most easily realized using self-checking pair (SCP) redundancy techniques to immediately stop a transmission if the output does not correspond with the data to be sent. An SCP approach could also be used at the receiver end. In addition to contributing to minimize the error-detection latency, such modular redundancy approaches would also contribute to higher error-detection coverage. In the actual DL designs, however, hardware redundancy is not used, as it is considered excessive for the intended application of the DLs and appropriate only in a context of node-level design redundancy. Instead, all error detection is done at the receiver exploiting the knowledge of message formatting and information redundancy.

At the DL-user level, a transaction involves the transmission of a data packet consisting of one or more data words. A design goal is to handle error detection at the receiver on a per-transaction basis ensuring no interference between transactions and with a small required gap between transactions in order to have as little impact on throughput as possible. To do this, a DL should have a design with cyclic operation that is fast enough to always complete the processing of a transaction, including all error detection, before the start of a new transaction. At the end of a processing cycle, the DL should be ready for another cycle with a properly initialized state. The critical design problem is how to do the processing with minimum delay and implementation size.

Two DL designs were developed for the SUT and the CFIMS. The RLs use a simple word-mode design in which each transaction handles a user-level message with one data word of predefined length specified at the time of system synthesis. The word-mode DL is a direct evolution of the design in [Torres08B] with enhanced error detection capabilities. The TCLs use a packet-mode design handling one or more data words per transaction as specified by the DL user on a per-transaction basis. The packet-mode DL is based on the word-mode DL design with added functionality to handle multiple data words as a single user message. The packet-mode DL design is supported by a multiple-access controller to arbitrate shared-link access by multiple sending processes.

6.1. Word-Mode Communication Unit

A Word-Mode Communication Unit (WMCU) includes a transmitter, a waveform monitor and a receiver. A word-frame consists of a sync pattern followed by the serialized user data followed by a CRC computed over the used data bits. Figure 6.3 shows a block diagram for the Word-Mode Transmitter. The design differs from the one in [Torres08B] in the use of a CRC generator to replace the simpler and less effective parity-bit generator. Also, after every word frame, the transmitter automatically inserts an idle pattern of sufficient length to accommodate the extra time needed at the receiver to complete a CRC check.

The design of the WMCU Waveform Monitor is unchanged from the one in [Torres08B]. The main purpose of this monitor is to reduce the likelihood that, due to the sampling operation of the receiver and the concomitant limited signal observability, an incoming random or highly distorted input signal may be interpreted as a valid word frame simply by chance. This waveform monitor tracks the input signal at the receiving end to detect timing envelope violations. A valid DL signal has two types of signaling intervals: sync coding and Manchester coding. A Manchester signaling interval includes the data and CRC sections of a word frame, and also the idle interval between word frames when a clock signal is transmitted. We define two valid waveform symbols in a sync coding interval: S0 and S1. Likewise, there are two valid Manchester coding waveform symbols: D0 and D1. Table 6.1 describes the symbols. Figure 6.4 illustrates the state transitions for the waveform monitor. After detecting a valid S0 symbol, the monitor follows the input signal as long as it satisfies the timing envelope constraints given by the valid symbol transitions. The signal is declared invalid immediately upon the detection of an envelope violation. The signal is declared valid only after a sufficient number of consecutive valid symbols have been received. [Torres08B] provides additional details about the design of the Waveform Monitor.

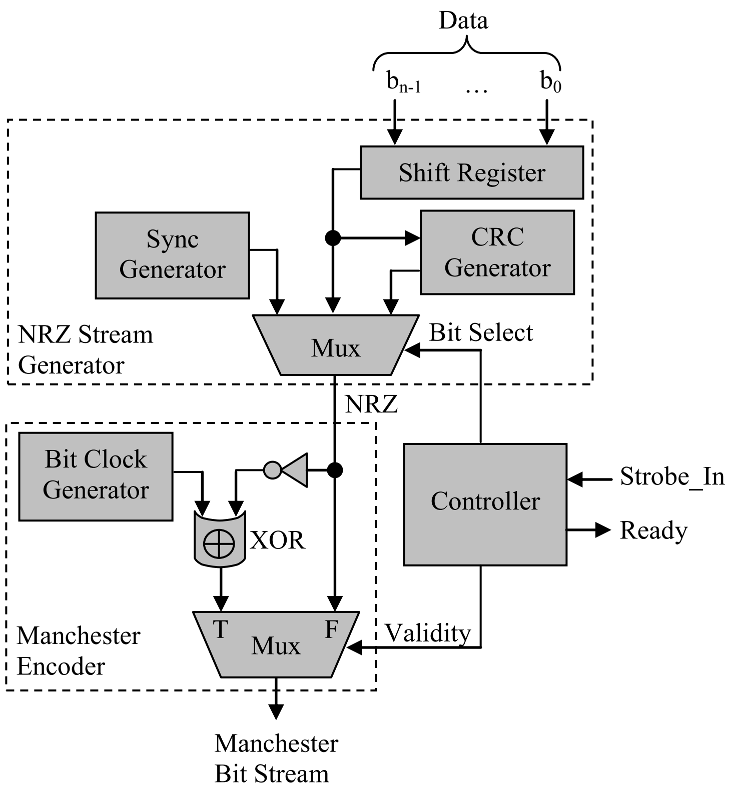


Figure 6.3: Block diagram for the word-mode transmitter

Table 6.1: Definition of WMCU waveform symbols

Signal Encoding	**Symbol**	**Signal Level**	**Nominal Level Duration (in half-bit time units)**
Sync	S0	0	3
Sync	S1	1	3
Manchester	D0	0	1-2
Manchester	D1	1	1-2

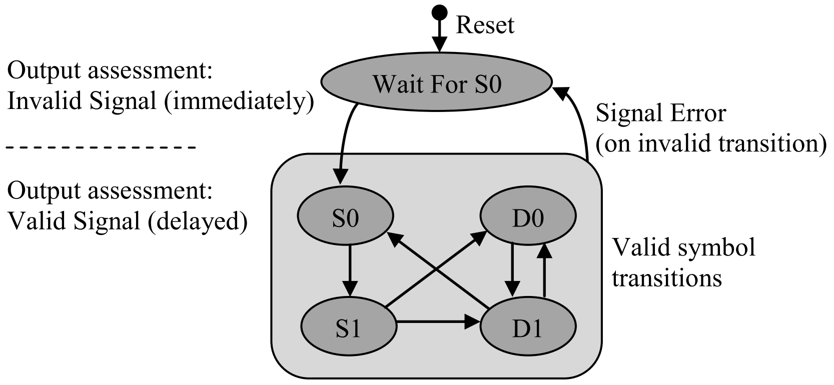


Figure 6.4: Simplified state diagram for WMCU Waveform Monitor

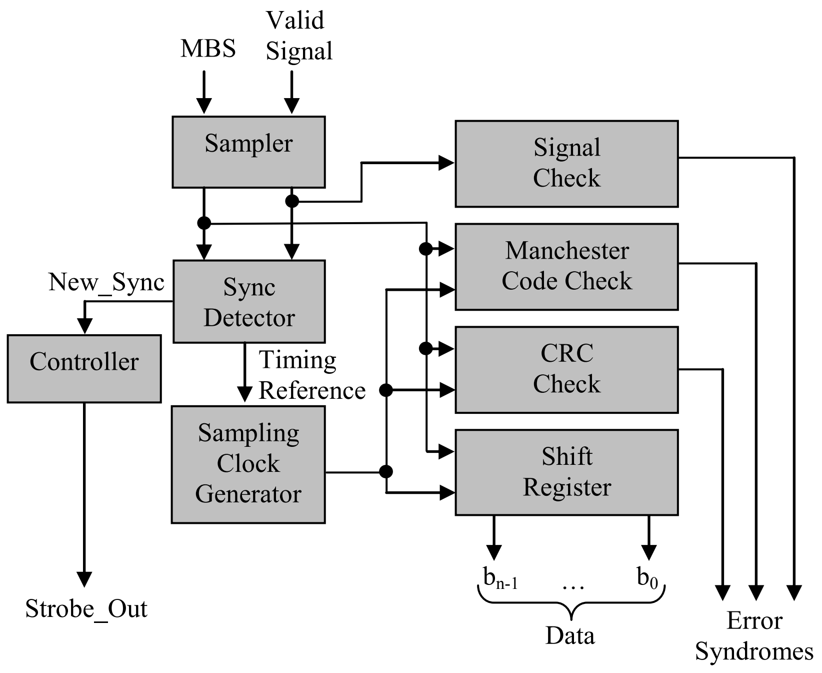


Figure 6.5: Block diagram for the Word-Mode Receiver

Figure 6.5 shows a block diagram for the Word-Mode Receiver. The Manchester-encoded Bit Stream (MBS) input is the signal received from the transmitter. The MBS signal is inspected by the Waveform Monitor before forwarding it to the receiver module. Valid Signal is generated by the Waveform Monitor. The Sampler block buffers the input signals long enough to resolve any possible metastable states. The Sync Detector examines the MBS input searching for a valid sync pattern. If one is found and Valid Signal remains asserted, a timing reference signal is generated and the Controller activates loading the remainder of the word frame. The Sync Detector is deactivated until the word frame processing is complete. Irrespective of any detected error, once New_Sync is asserted, the receiver will generate an output with data and error syndromes and a time delay determined by the word frame length. If Valid Signal is deasserted at any time during the word frame, the corresponding error output (or "syndrome") will be asserted. The Manchester Code Check is applied to the data and CRC parts of the word frame, and an error is reported if there is code violation. The CRC Check compares the received CRC bit string with the locally computed CRC. The Strobe-Out signal is asserted once after the word frame processing is complete. The receiver then returns to the ready state and the sync Detector is reactivated.

6.2. Packet-Mode Communication Unit

A Packet-Mode Communication Unit (PMCU) consists of a packet-mode transmitter, a waveform monitor and a receiver. The purpose of the PMCU is to transfer data packets (i.e., blocks of data) from a sending process to a receiving process in a single transaction. Henceforth, the sending and receiving processes are referred to as the (link) users. The size of the data packet is specified by the sending process at the beginning of a transaction. As stated previously, the packet-mode DL is based on the word-mode DL with expanded functionality to handle blocks of user data as a single message. Figure 6.6 illustrates the packet format used by the PMCU. The packet header has a user-specified tag and the length of the payload section, which is denoted by L. The tag field value is meaningful only to the users and, relative to the PMCU operation, it is simply another user data item. The packet length is sufficient information to properly receive a packet. Because of the critical importance of the packet length, the integrity of the packet header is protected by its own CRC-based FCS, referred to as the header FCS (HFCS). The payload section has L user data words as indicated in the header. The trailer is the payload FCS (PFCS) consisting of a CRC code computed over the content of the packet payload section.

Header	Tag	Payload Length (L)	Header FCS
Payload	Payload Word 0		
	● ● ●		
	Payload Word L-1		
Trailer	Payload FCS		

Figure 6.6: PMCU packet format

Figure 6.7 illustrates the format for a packet frame, which consists of a sequence of word frames with the packet content. Notice the use of the Sync1 pattern to mark the word frame carrying the packet header.

L+2 word frames	Sync1	Packet Header
	Sync0	Payload Word 0
	● ● ●	● ● ●
	Sync0	Payload Word L-1
	Sync0	Packet Trailer

Figure 6.7: Packet frame format

Figure 6.8 shows a high-level block diagram for a packet-mode data link using PMCU components. The Packet Frame Encoder (PFE) serves as a user interface and generates a packet with null FCS fields (see Figure 6.6). The Word Frame Encoder (WFE) is a modified version of the NRZ Stream Generator block in the WMCU transmitter (see Figure 6.3) with the capability to generate Sync0 and Sync1 patterns and the CRCs for the HFCS and PFCS of a packet. The WFE output is a bit-serial, NRZ-encoded packet frame as in Figure 6.7. The PFE generates the packet words at a predetermined constant rate that is sustainable by the WFE to generate the word frames. The Bit Encoder is identical to the Manchester Encoder in Figure 6.3.

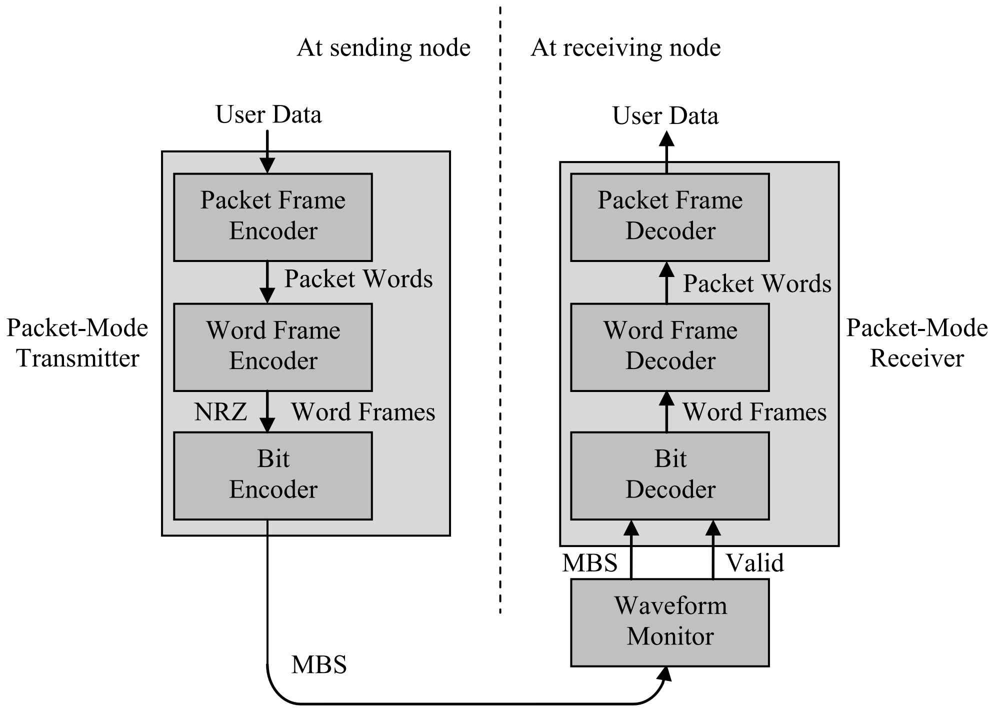


Figure 6.8: Block diagram for the packet-mode data link

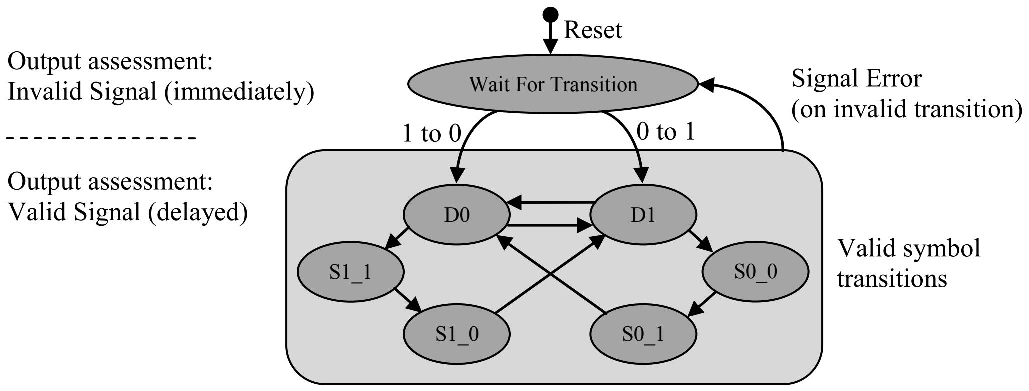
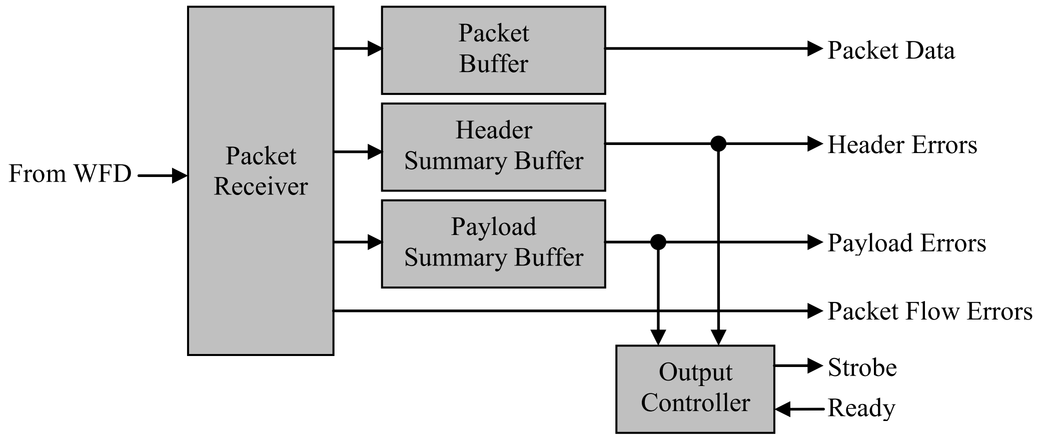


Figure 6.9: Simplified state diagram for PMCU waveform monitor

The PMCU design expands the WMCU Waveform Monitor design to accept Sync1 patterns. Figure 6.9 is a simplified state transition diagram for the Packet-Mode Waveform Monitor. Now, symbols S0_0 and S0_1 denote Sync0 levels 0 and 1, respectively. Likewise, symbols S1_1 and S1_0 respectively denote Sync1 levels 1 and 0. The duration constraints for Sync1 are the same as for Sync0 and are given in Table 6.1. Notice that, similarly to the WMCU Waveform Monitor, the PMCU Waveform Monitor does not incorporate constraints for the minimum separation between consecutive valid sync patterns. This is a deliberate omission to limit the design complexity. That error check capability can be easily added if deemed sufficiently beneficial in terms of increased error coverage.

In Figure 6.8, the purpose of the Bit Decoder is to detect valid sync patterns and generate the sampling clock reference used by the Word Frame Decoder (WFD). The WFD deserializes the word frames and computes the HFCS for received header words and the PFCS for packet payloads. The WFD also performs signal validity and Manchester encoding checks on every received word frame.



Header Errors:	Payload Errrors:	Packet Flow Errors:
- SD Error	- SD Error	- Unexpected Header
- MC Error	- MC Error	- Unexpected OTH
- CRC Error	- CRC Error	- Packet Buffer Unavailable
	- Length Error	- Summary Buffer Unavailable
	- Timing Error	
	- Buffer Overflow	

Figure 6.10: Block diagram for the Packet Frame Decoder and list of packet error checks

The purpose of the Packet Frame Decoder (PFD) is to organize the words from the WFD into valid packets and buffer the packets until the user is ready to read them. Figure 6.10 shows a simplified block diagram for the PFD and the list of defined packet error checks. The Packet Receiver stores received words in the Packet Buffer. The Header and Payload Summary Buffers hold the error syndromes for received packet headers and payloads. The Packet Receiver is designed such that, for every Header Summary Buffer entry, there is a corresponding Packet Summary Buffer entry and a corresponding Packet Buffer entry consisting of one of more words, including a packet header. The Output Controller manages reading of the buffers based on the content of the Summary Buffers and the readiness of the user to accept received packets. The Packet Receiver performs a significant number of checks, as can be seen by the listed error outputs in Figure 6.10. The Header Error checks are the same performed by the WMCU receiver for a generic word frame. The SD (Signal Detect, or Valid Signal, as generated by the Waveform Monitor) and MC (Manchester Code) checks are applied to every payload word, including the PFCS. An error in any of them invalidates the whole packet. The Payload CRC Error is based on the result of the PFCS check. The Length Error indicates that the packet was rejected due to the arrival of a new header word before the PFCS for the current packet was received. The Timing Error is based on a timeout check for the gap between packet words, which can be bounded using the known word generation rate at the transmitter. The Buffer Overflow error occurs when the Packet Buffer becomes full before all the words of an apparently valid packet being received have been stored. The Packet Flow Errors are reported by the Packet Receiver directly to the user interface as they can occur at any time and can affect the ability to use the buffers. These errors are assumed to be due to a break-down in packet transfer coordination between the transmitter and the receiver, or between the receiver and the user. The Packet Receiver rejects incoming packets as long as any of the three buffers is full. If a Packet Buffer overflows, the Packet Receiver stalls and rejects incoming words until the Output Controller has flushed out all the buffers.

The Output Controller uses the error syndromes stored in the Summary Buffers to determine how to process the content of the Packet Buffer. If a Header Error is asserted, it is assumed that the Packet Receiver stored the packet header but aborted the process of receiving and loading the packet payload onto the buffer. In this case, the Output Controller outputs the received header as stored in the buffer, together with all the corresponding header and payload errors in the Summary Buffers. If there are no Header Errors but at least one Payload Error is asserted, the Output Controller outputs the packet header and the error syndromes, and then discards any part of the payload stored in the Packet Buffer. If none of the Header or Payload Errors is asserted, the full packet content is forwarded to the user.

6.3. Multiple-Access Controller

The PMCU Multiple-Access Controller allows multiple sending processes to share a data link. Figure 6.11 is a block diagram of the Multiple-Access Controller. The signals in Figure 6.11 are prefixed to identify the source or sink as either the transmitter (prefix "Tx_") or the user processes (prefix "User_"). Simple priority was chosen for the access policy as it is well suited for the intended use in which there is clear precedence among the sending processes (e.g., round control, function testing, and fault injection processes, in that order, at the PTC). To request access to the transmitter, a user asserts its User_RTS signal (RTS = Request To Send) and waits for the corresponding User_CTS signal (CTS = Clear To Send). When the transmitter is available, as indicated by signal Tx_Ready, the controller gives access to the waiting user of highest priority by asserting the corresponding User_CTS, selecting the proper port at the multiplexers and starting a transmitter transaction (i.e., a data packet transmission) by asserting Tx_Strobe. The selected user retains access to the transmitter for one transaction, at the end of which Tx_Ready is reasserted and access is re-evaluated.

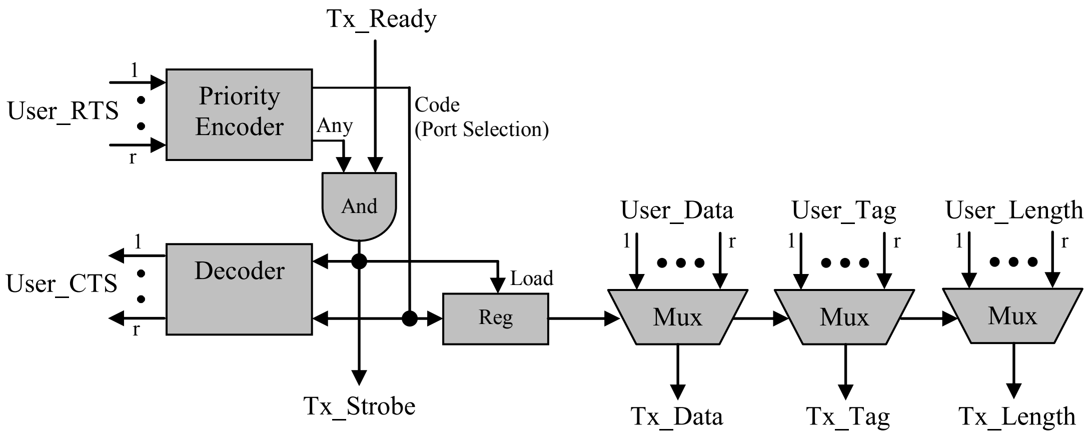


Figure 6.11: Block diagram for the PMCU Multiple-Access Controller

7. Test Control Messages

The Test Control Links (TCLs) are a point-to-point communication infrastructure that enables message-based interaction between CFIMS hardware processes running on different nodes at the Test Controllers and the SUT (see Section 4.3). The functional-level messages exchanged by hardware processes over the TCLs are called Test Control Messages (TCMs).

Figure 7.1 shows the format of a TCL packet carrying a TCM. In the current version of the system, a TCL packet is a series of 16-bit data words with a packet header, a set of one or more payload words and a packet trailer. The TCM content includes the message source id, a tag indentifying the nature of the TCM, and W 16-bit payload words. The bits to the right of the TCM tag field may also be used to carry context-dependent TCM payload data. The TCM sources in the current version of the system are listed in Table 7.1. Table 7.2 lists the TCM tags, which identify the content of the TCMs. In general, the format of the TCMs is specific to the message context and varies with the TCM tag. The following subsections describe the content and format of the TCMs.

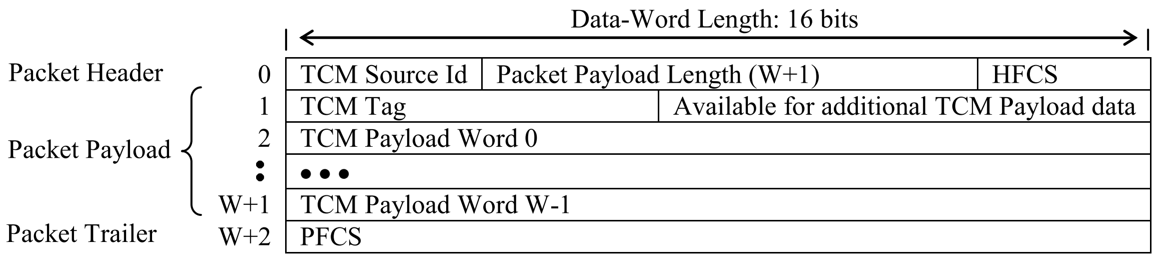


Figure 7.1: TCL packet format for TCMs

Table 7.1: TCM Sources

Source Node	Source Process	Source Id
PTC	Round Controller (RCtlr)	PTC_ROUND_CONTROLLER
	Function Timer (FTmr)	PTC_FUNCTION_TIMER
	Function Tester (Function Monitors, FMons)	PTC_FUNCTION_TESTER
	Fault Injection Controller (FIC)	PTC_PE_BIU_FIC
STC	Round Controller (RCtlr)	STC_ROUND_CONTROLLER
	Fault Injection Controller (FIC)	STC_RMU_FIC
PE-BIU	Processing Element (PE)	PE_BIU_PE
	Embedded Node Monitor (ENM)	PE_BIU_ENM
RMU	Embedded Node Monitor (ENM)	RMU_ENM

Table 7.2: TCM Tags

Source Processes	TCM Tag	Description
PTC and STC Round Controllers	RC_ENABLE	CCP message used to signal that the controller sending the message has been enabled by the software to begin a round
	RC_READY	CCP message used to signal that the controller sending the message has determined that SPIDER has been successfully initialized
	RC_START	CCP message used to signal that the controller sending the message has completed the round setup procedure
	RC_STOP	CCP message used to signal that the controller sending the message is in the process of stopping the execution of the round
PEs	SYNC_PE_TIME	PE synchronization message
FTmr	SYNC_ROUND_TIME	PE synchronization message; also used to synchronize CFIMS processes
Function Tester (FMons)	SF_SETUP	Function setup message sent to the PE-BIU nodes during the CCP Function Setup mode
	SF_EXECUTE	Function enable sent to the PE-BIU nodes during the CCP Function Execution mode
PEs and FMons	SF_DATA	Application data exchanged by the PEs and the FMons
PTC FIC (relayed to RMU nodes by STC FIC)	FI_RESET	Commands a reset of the fault injectors
	FI_FAULT	Fault-injector setup command sent during the CCP Fault Injection Setup mode
	FI_ACTIVATION	Fault-injector setup command sent during the CCP Fault Injection Setup mode
	FI_EXECUTE	Fault-injector enable command
	FI_FIRE	Fault injection reference event
PE-BIU ENM	SM_PE_BIU	Sample of PE-BIU node state
RMU ENM	SM_RMU	Sample of RMU node state

7.1. Round Control

The round control TCMs are exchanged between the PTC and STC Round Controllers during the execution of the CCP (see Section 5). The Enable message (TCM tag: RC_ENABLE) is used in the System Enable major mode. Figure 7.2 illustrates the format for this message. The values for the Round Index and Enabled Nodes fields, which are, respectively, the unique identifier for the round and a runtime configuration vector specifying the active nodes for the round, are supplied by the PTC software at the beginning of the round. Both fields have arbitrary values in the initial CCP message from the STC to the PTC.

	b_1 •••	b_{16}
1	Tag: RC_ENABLE	--- Unused ---
2	Round Index	
3	Enabled Nodes	

Figure 7.2: TCM format: RC_ENABLE

Note that in Figure 7.2 the TCM fields are in the same relative location as in the actual implementation, but the field lengths are not proportional to the actual field lengths. This is also true in the other TCM format figures.

The Ready and Start messages do not carry any additional data beyond the TCM tag. Figure 7.3 shows the format for these messages.

	b_1	••• b_{16}
1	Tag: RC_READY or RC_START	--- Unused ---

Figure 7.3: TCM format: RC_READY and RC_START

The Stop messages help return the system to the idle mode in a coordinated manner. A Stop message includes a field that specifies the Round Controller's stop condition, which is reported to the software at the PTC and STC when the round execution has fully stopped. Figure 7.4 shows the format for the RC_STOP TCM message. A full list and description of the stop conditions is given in Appendix B.

	b_1	••• b_{16}
1	Tag: RC_STOP	Stop Condition

Figure 7.4: TCM format: RC_STOP

7.2. Function Testing

Function testing includes setup and execution for PE synchronization and SUT application testing. Figure 7.5 shows the format for the SUT setup message, which includes all the runtime configuration information needed by the PEs. The Enabled Nodes field specifies the active nodes for the round. The Application Assignment is a vector that specifies the function to be performed by each active PE as either Input-Output or Control (see Section 4.3.1.2). The next field specifies the number of ROBUS Cycles (RCs) per Control Cycle (CC), which specifies the duration of a control cycle. (Note that the duration of a ROBUS Cycle is fixed at synthesis time.) The last series of words are the PE communication schedule. As stated in Section 4.3.1.1, the PE communication schedule for a control cycle is defined as an indexed set of ROBUS communication schedules, with one ROBUS schedule for each ROBUS cycle in a control cycle, and with the schedule index identifying the ROBUS cycle in which to apply a particular ROBUS schedule.

	b_1	••• b_{16}
1	Tag: SF_SETUP	--- Unused ---
2	Enabled Nodes	
3	Application Assignment	
4	RCs per CC	
5	PE Schedule: Word 0	
⋮	•••	
S + 4	PE Schedule: Word S-1	

Figure 7.5: TCM format: SF_SETUP

The Execute message is sent to the PE-BIU nodes at the beginning of the CCP Function Execution major mode (see Section 5) to enable the PEs. The TCM format is given in Figure 7.6.

	b_1	••• b_{16}
1	Tag: SF_EXECUTE	--- Unused ---

Figure 7.6: TCM format: SF_EXECUTE

The following subsections cover the TCMs for execution of the PE synchronization protocol and SUT application testing.

7.2.1. PE Synchronization

The PE synchronization protocol is described in Section 4.3.1.1. Two types of messages are involved. The messages from the PEs to the FTmr include the RCI at the PEs. The messages from the FTmr to the PEs and other CFIMS processes include the RCI and CCI values as determined by the FTmr. The formats for these messages are shown in Figures 7.7 and 7.8, respectively.

	b_1 •••	b_{16}
1	Tag: SYNC_PE_TIME	--- Unused ---
2	ROBUS Cycle Index (RCI)	

Figure 7.7: TCM format: SYNC_PE_TIME

	b_1 •••	b_{16}
1	Tag: SYNC_ROUND_TIME	--- Unused ---
2	ROBUS Cycle Index (RCI)	
3	Control Cycle Index (CCI)	

Figure 7.8: TCM format: SYNC_ROUND_TIME

7.2.2. Application Testing

The application data TCM from an FMon to a PE contains the sensor or command data to be transmitted by the PE over ROBUS as a single application-level message. In addition to the application data, the TCM includes a PE-message header with fields for the Id of the sender PE, an error flag to be asserted by the sender PE if it detects an error in the message from the FMon, and another error flag to be asserted by the receiving PE if it detects an error in the message from the sender PE over ROBUS. After the scheduled time to receive the message from the sender PE, the receiving PE sends to its corresponding FMon a TCM with its ROBUS communication observations (see Section 4.3.1.2). Figure 7.9 shows the format for an application testing data message.

		b_1	•••		b_{16}
	1	Tag: SF_DATA		--- Unused ---	
PE Message	2	PE Sender Id	Error Detected at Sender PE	Error Detected at Receiver PE	--- Unused ---
PE Message	3	Application Data Word 0			
PE Message	⋮	•••			
PE Message	D+2	Application Data Word D-1			

Figure 7.9: TCM format: SF_DATA

7.3. Fault Injection

The Fault Injection Controller in the PTC (see Section 11.3) builds each of the fault injection TCMs to control the operation of the Local Fault Injectors (see Section 8). The messages are sent to the PE-BIUs from the PTC Fault Injection Controller by means of the PTLs. The Fault Injection Controller also sends the messages along the CCL to the STC, where the TCMs are forwarded to the RMUs by means of the

STLs. The format of the first payload word of all the fault injection TCM packets is the same. Figure 7.10 shows the format of the first payload word of the TCL packet format for all TCMs given in Figure 7.1. Bits 1 through 4 are used to identify the tag of the TCM. The rest of the payload word is left for the fault injection destination. The destination address specifies the receiving Local Fault Injector. When the message is meant for all Local Fault Injectors, the fault injection destination can set the message as broadcast. Upon receiving a broadcast message, all of the Local Fault Injectors will process the fault injection TCM. To reach individual fault injectors, the Broadcast bit is cleared and the Fault Injector ID, Node Id, and Node Kind subsections of the destination are set to the address of the target fault injector.

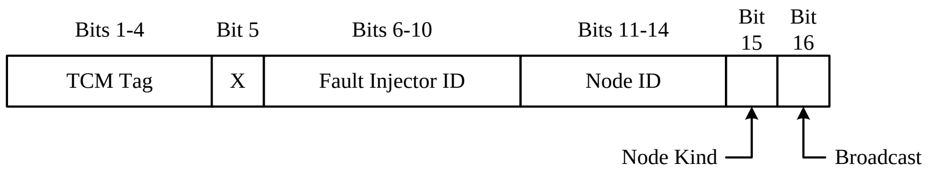


Figure 7.10: Format of first TCM Payload Word for FI_RESET, FI_FAULT, FI_ACTIVATION, FI_EXECUTE, and FI_FIRE

The FI_RESET TCM is broadcast to all Local Fault Injectors at the start of a round. The format of this message is shown in Figure 7.11. The second payload word contains the same Enabled Nodes field as the RC_ENABLE TCM (see Section 7.1). The Local Fault Injectors use this field to determine when they are located in an active node.

	b_1 •••	b_{16}
1	Tag: FI_RESET	FI Destination
2	Enabled Nodes	

Figure 7.11: TCM format: FI_RESET

The FI_EXECUTE message format shown in Figure 7.12 only contains the TCM tag. All Local Fault Injectors receive this message by means of the Broadcast field. This TCM indicates the start of the Execution mode for the Local Fault Injectors (see Section 8.1).

	b_1 •••	b_{16}
1	Tag: FI_EXECUTE	FI Destination

Figure 7.12: TCM format: FI_EXECUTE

Figure 7.13 gives the format of the Fault, Activation, and FIRE messages. These TCMs can be applied to all of the Local Fault Injectors in the SUT by using the Broadcast field in the destination section of the data word. Alternatively, multiple Fault, Activation, and FIRE TCMs can be used to control faults at different Local Fault Injectors. The Fault or Activation ID in the payload of the TCM indicates the fault type and activation pattern (see Section 4.3.2), respectively, whereas the FIRE ID distinguishes which reference event has occurred. The multiplexer inputs chosen define what fault type or activation pattern to set up for the test. Any subsequent payload data words in the TCM are optional parameters relating to the particular fault type chosen.

	b1 •••	b16
1	Tag: FI_FAULT, FI_ACTIVATION, or FI_FIRE	FI Destination
2	Fault or Activation or FIRE ID	
3	Parameter: Word 1	
⋮	•••	
P + 2	Parameter: Word P	

Figure 7.13: TCM format: FI_FAULT, FI_ACTIVATION, and FI_FIRE

In the current version of the CFIMS, the only FI_FIRE TCM has a FIRE ID of 0. The format of this message is given in Figure 7.14. This reference event is used to specify the faulty nodes for the control cycle and remove these faults at the end of the control cycle, respectively, by means of the Fault Vector and Disable Vector field (see Section 8.2).

	b1 •••	b16
1	Tag: FI_FIRE	FI Destination
2	FIRE ID	
3	Fault Vector or Disable Vector	

Figure 7.14: TCM format: FI_FIRE for current CFIMS

7.4. State Monitoring

The collection of SUT node state data is described in Section 4.3.3. Basically, an Embedded Node Monitor (ENM) gathers state data from its local node and sends it via the TCL to its corresponding State Monitor process at a test controller. There are various transmission triggers programmed into the ENMs, including local RPP mode transitions and failure detection events. Figure 7.15 shows the TCM format for a PE-BIU state message. The Sequence Number field is an 8-bit message count intended to help identify the loss of transmitted state messages. The Trigger Id field identifies the condition that triggered the transmission of the state message. The ROBUS Links (RLs) Status indicates whether any of the RLs detected a received-signal error (as reported by the WMCU Waveform Monitor of the RL) at any time since the previous state message was transmitted. The remainder of the TCM includes snapshots of RPP and PE state variables. The TCM format given in Figure 7.16 for RMU state messages is the same as for PE-BIU state TCMs except for the absence of PE data. Appendix B provides more detailed information about the content and format of the SUT state monitoring messages.

		b1 •••	b16
	1	Tag: SM_PE_BIU	Sequence Number
	2	Trigger Id	RLs Status
RPP (BIU)	3	RPP: MCU Command	
	4	RPP: Accusations	
	5	RPP: Convictions	
	6	RPP: SMU State Sample	
PE	7	PE: State sample	
	8	PE: RCI	

Figure 7.15: TCM format: SM_PE_BIU

		b_1 •••	b_{16}
	1	Tag: SM_RMU	Sequence Number
	2	Trigger Id	RLs Status
RPP (RMU)	3	RPP: MCU Command	
	4	RPP: Accusations	
	5	RPP: Convictions	
	6	RPP: SMU State Sample	

Figure 7.16: TCM format: SM_RMU

8. Fault Injectors

To begin the design of the Local Fault Injectors, there must be a clear definition of where faults can exist and what type of faults can be injected. Since the goal of the fault injection experiment is to characterize faults in the ROBUS-2 system (see Section 4.1), which contains BIUs and RMUs, fault injections in the system are only permitted within these regions. Each fault injector module operates on a single-bit signal within the component. In order to be able to study various faults, the fault injector architecture needs to be easily extended to include multiple fault types and activation patterns of these faults.

8.1. Common Architecture

Each fault injector has a common internal structure for reusability in each component as well as on different signals within a component. In order to induce faults in specific fault injectors, each injector is individually addressable by the top-level Fault Injection Controller. The high level state machine of the Local Fault Injector main controller is shown in Figure 8.1. The primary objective is to screen all incoming TCMs and store the messages that apply to the fault injector. The controller checks whether the TCM tag is one that applies to the Local Fault Injectors and if the message is meant for this particular fault injector. For more information on the TCM tags see Section 7. When a Reset message (TCM tag: FI_RESET) is received, the main controller sets the reset signal. The message contains the Enabled Nodes vector that specifies active nodes for the round from the Test Specification file (see Appendix A). In the case where the node is enabled, the reset signal is then disabled to allow the Local Fault Injector to operate on the signal that is susceptible to faults. Upon receiving an Execute message (TCM tag: FI_EXECUTE), the Execute flag is set in the register by the main controller allowing the activation pattern selected for the fault to be applied to the Injection Multiplexer. Signals from the main controller of the Local Fault Injector and the Packet Mode Communication Unit Receiver are sent through various delays and logic to determine when to write to the Fault, Activation, Sync, and each Fault Injection Reference Event (FIRE) buffer. See Section 7.3 for the format of each fault injection TCM. The fault and activation buffers are used to store the selection signal used to choose the appropriate input of the corresponding multiplexer. These buffers also have the ability to store additional parameters that affect the operation of the input function. Setup parameters and reference events can be sent to particular fault types and activation patterns to control their behavior. Consequently, the FIRE buffers were built in the design to hold parameters of these events. The Sync message (see Section 4.3.1.1.) can also trigger a specific behavior within these inputs, so the RCI and CCI are stored in the Sync buffer.

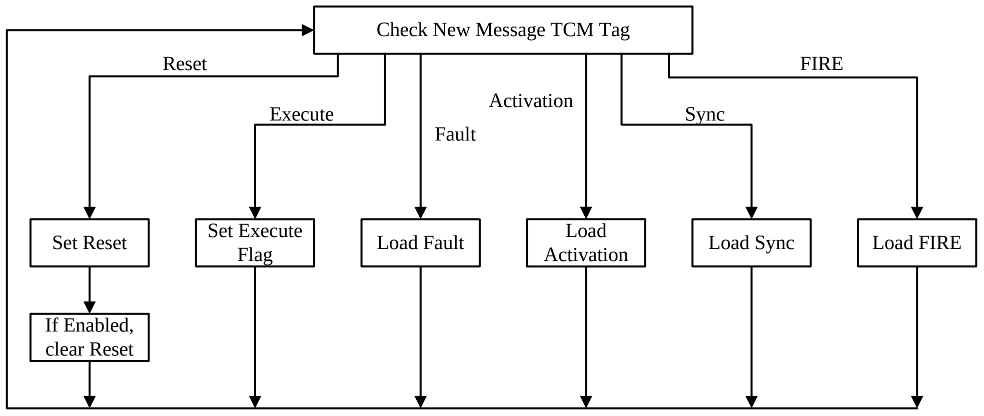


Figure 8.1: Local Fault Injector State Machine

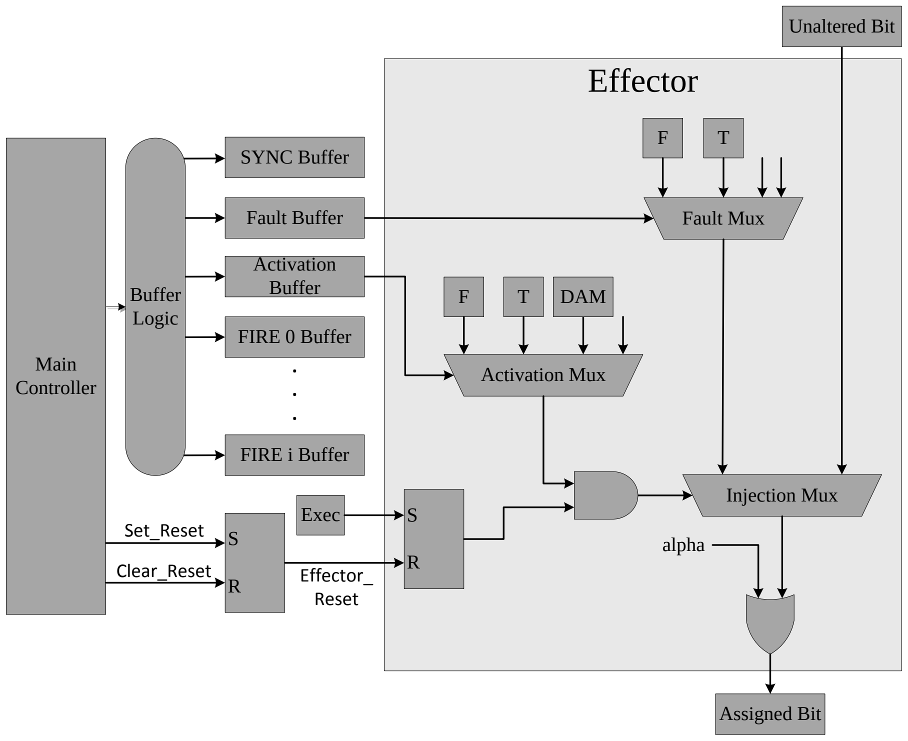


Figure 8.2: Local Fault Injector Architecture

The general architecture of the Local Fault Injector is shown in Figure 8.2. The section of the Local Fault Injector that alters the signal is called the Effector. Each input signal of the Activation Multiplexer and the Fault Multiplexer can be changed to obtain the desired fault types. The Unaltered Bit is the signal that is to be affected and the Assigned Bit is either the altered signal through the Fault Multiplexer or the actual signal value from the Unaltered Bit input. The switching between the unaffected and faulty signal is determined by the activation pattern from the Activation Multiplexer as long as the injector is in Execute mode and the Effector is not reset. When the signal susceptible to fault injection is the reset signal of the node, the Alpha signal is connected to the hardware reset. When the hardware reset is set, it takes precedence over the signal from the Effector. However, when the Local Fault Injector is not altering the reset signal, the Alpha signal is set to zero to allow the output of the Injection Multiplexer to pass through as the Assigned Bit.

8.2. Fail-Silent Fault Injector

To implement fail-silent type faults (see Section 4.3.2) the Local Fault Injector with an address of 0 is placed on the functional Reset signal of the SPIDER nodes. When a fault occurs on a node, the Reset signal of the node is set so that the node resets and does not perform any function. This application of the Reset signal satisfies the fail-silent definition because the node does not produce any outputs. When the node is reset for any reason, this signal must override any altered signal of the fault injection. Therefore, the Alpha signal is connected to the Effector's Reset signal allowing the node to be reset and bypass the functionality of the Effector.

The first and second inputs of both of the Fault and Activation Multiplexers are set to a constant low and high signal, respectively. For this CFIMS, the Direct Activation Mode (DAM) function was added as a third input to the Activation Multiplexer. The activation value associated with the DAM is affected by the reference event and parameters corresponding to FIRE 0. At the beginning of a round, the fault type value is sent to the buffer to select the input that is always high on the Fault Multiplexer and the activation value is sent to its buffer to select the DAM input. These values stay in the buffers throughout the remainder of the round. When it is time within a control cycle, a FIRE TCM with the ID of 0 and the fault vector (see Section 4.3.2) for the control cycle is sent to all of the SPIDER nodes through a broadcast destination address. The DAM is notified when a new value has been written to the FIRE 0 buffer and the DAM finds the value in the fault vector corresponding to the ID of the node where the injector is located to see whether or not the node is supposed to be upset for that control cycle. When the node is supposed to be upset, the output of the Activation Multiplexer selects the output value of the Fault Multiplexer. However, when the node is not defined as upset in the fault vector, then the Activation Multiplexer chooses the Unaltered Bit, which is the value that has not been tampered with for the output functional reset value. Toward the end of the control cycle, another FIRE TCM is sent with an ID of 0 and a fault vector with all values cleared to disable the fault. Similar to the previous FIRE message, the DAM obtains the value within the cleared fault vector associated with the node where the fault injector is located from the FIRE 0 buffer. This bit within the fault vector is never set, so the output of the Activation Multiplexer always chooses the Unaltered Bit value to send on to the output of the Effector. This process is then repeated for each control cycle throughout the execution phase of the test.

9. PE-BIU Node

At the functional level, the SUT consists of the ROBUS communication components (i.e., BIUs and RMUs) and the application-level computation components (i.e., the PEs). At the physical level, the SUT is implemented with PE-BIU physical nodes that execute PE and BIU functions, and RMU nodes executing the RMU function. This subsection describes the function and implementation of a PE-BIU node. The next section describes the RMU node.

In addition to implementing the PE and BIU functions, a PE-BIU node must include functionality for fault injection, state monitoring and runtime configuration. Section 4.3 presents an overview of these system design aspects. The PE-BIU node must also be able to interact with the PTC controller, which executes a round according to the CCP protocol described in Section 5. Due to the limitations of the RSPP1 execution nodes, all the PE-BIU node functionality in the current design is implemented as hardware processes running on an FPGA.

Figure 9.1 shows a functional block diagram for a PE-BIU node. The components are divided into SUT functional processes (i.e., processes performing SUT-specific functions) and CFIMS processes that support the functions of configuration, application testing, fault injection and node monitoring. All of these modules run on the FPGA of an RSPP node programmed as a PE-BIU node. The RPP configured for BIU functionality is the central SUT component of a PE-BIU functional node. The design and implementation of the RPP is described in [Torres05B]. The RPP was modified as described in [Torres08B] to support state monitoring. The Node Id is manually programmed onto the flash memory of the FPGA board (see Section 4.2.1) and is given to the RPP by a support module (not shown) immediately after the FPGA is programmed. The WMCU components connected to the BIU handle the communication with the RMU nodes (i.e., transmission and reception of ROBUS messages). All the ROBUS-specific processing is handled by the RPP.

The main function of the PE module is to relay application-level messages between ROBUS and FMons at the PTC. For this, the PE must coordinate its operation with the local BIU and with other PEs via the FTmr. The round-specific runtime configuration of the PE is handled by the PE Setup Unit, which receives the setup information sent by the corresponding FMon during the CFIMS setup phase. The design of the PE Setup Unit and the PE are described in the following subsections.

The PE and the BIU monitor their own operation and the operation of the system by means of local error checks. If either of these modules detects a local- or SPIDER-level failure condition, the module halts its execution and asserts an output signal to report the condition. As shown in Figure 9.1, these PE and BIU failure signals are connected to a reset circuit whose function is to restart all the SUT functional node modules, thus eliminating any persistent node state errors (assuming the faults are transient).

The output of the Reset module is routed through Fault Injector 0 to enable the simulation of a PE-BIU fail-stop failure mode. Other optional fault injectors may be added to simulate particular faults and observe the node response, or to force particular DCP failure modes (see Section 2) to observe the system-level response.

The ENM module collects state information from the node components and packages it onto TCMs that are transmitted to the PTC State Monitor (SMon) upon the occurrence of a trigger condition. The design of the ENM is described in a later subsection.

The PMCU is shared by multiple modules for transmission and reception. The PE and the ENM share the transmitter, with the PE connected to the highest priority port in the multiple-access controller. The receiver is shared by the PE, the PE Setup Unit and the Fault Injectors.

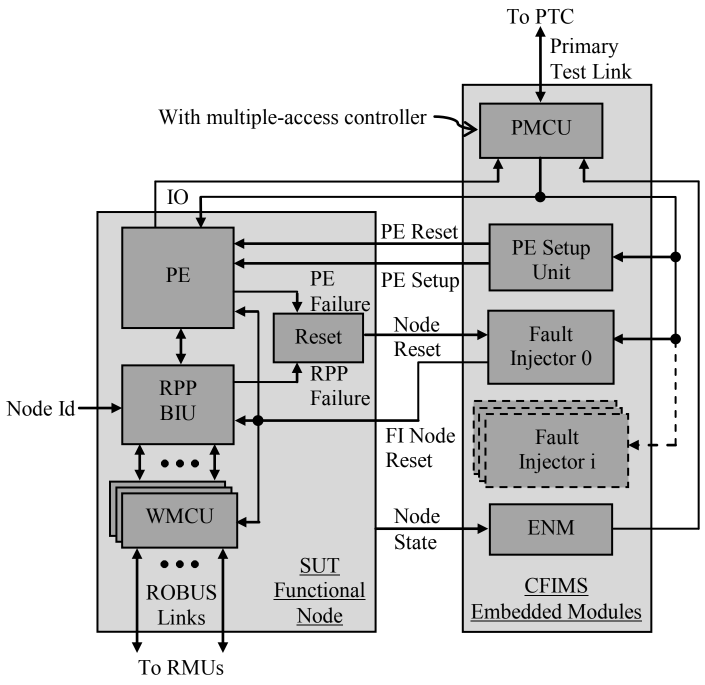
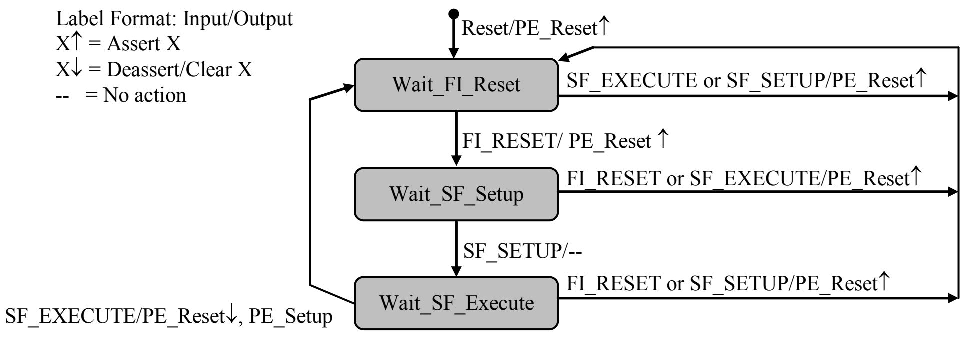


Figure 9.1: Block diagram for a PE-BIU node

9.1. PE Setup Unit

The PE Setup Unit (PESU) is intended to complement the functionality of the PE by handling all the interaction with the CFIMS prior to the execution phase of a test round. In particular, the PESU is responsible for directly disabling the PE during the SPIDER Initialization CFIMS mode, configuring the PE during the Function Setup mode, and enabling the PE at the beginning of the Function Execution mode.

Figure 9.2 shows the mode transition diagram for the operation of the PESU. The PESU accepts TCMs with three different tags: FI_RESET, SF_SETUP and SF_EXECUTE. TCMs with other tags are ignored. A local FPGA reset triggers the PESU to disable the PE and wait for an FI_RESET TCM, which is sent by the PTC Fault Injection Controller at the beginning of the SPIDER Initialization mode to trigger a re-initialization of the SUT. After that message is received, the PESU expects the arrival of PE runtime configuration data in an SF_SETUP TCM. The PE_Reset signal remains asserted by the PESU until the arrival of the SF_EXECUTE message, which signals the transition of the CFIMS to the Function Execution mode. At that point in time, the PE is released to execute its function. If the PESU receives a message out of order, it immediately reasserts the PE Reset signal and restarts the process.



Label Format: Input/Output
X↑ = Assert X
X↓ = Deassert/Clear X
-- = No action
Reset/PE_Reset↑
Wait_FI_Reset
SF_EXECUTE or SF_SETUP/PE_Reset↑
FI_RESET/ PE_Reset ↑
Wait_SF_Setup
FI_RESET or SF_EXECUTE/PE_Reset↑
SF_SETUP/--
Wait_SF_Execute
FI_RESET or SF_SETUP/PE_Reset↑
SF_EXECUTE/PE_Reset↓, PE_Setup

Figure 9.2: Mode transition diagram for the PE Setup Unit

The configuration data carried by the SF_SETUP TCM (see Section 7.2) includes the selection of which SUT nodes will be active during the round, the function to be performed by each active PE (i.e. IO or Control), the duration of a control cycle in terms of the number of ROBUS cycles, and the PE communication schedule to be used in every control cycle. The PESU performs a simple validity check for each of the first three configuration parameters: if any of them is all zeros (which means, respectively, that none of the PEs are active, all the active PEs are configured to perform IO functions, or the duration of the control cycle is zero), the configuration process is aborted and the PESU returns to waiting for an FI_RESET message while holding the PE is the reset state. The PE communication schedule at the control-cycle level is an indexed set of ROBUS-cycle-level communication schedules (see Section 3.7.2), with one ROBUS schedule for each ROBUS cycle in a control cycle, and with the schedule index identifying the ROBUS cycle in which to apply a particular ROBUS schedule. The PE communication schedule in the SF_SETUP TCM includes only the non-zero ROBUS communication schedules. The PE is responsible for generating a full control cycle communication schedule using this compressed schedule format. The PE also performs validity checks on the communication schedule after the reset signal is cleared.

9.2. Processing Element

The PE is the SUT component that performs the user-level application. Currently, this consists of bidirectional relaying of messages between the FMons and ROBUS. This user-level service is supported by platform-specific services implemented at the PEs and lower-level services provided by ROBUS. The PEs are fully implemented in hardware and each one shares an FCR with its attached BIU. There are two main interface ports on a PE: the ROBUS port for interacting with the BIU, and the IO port for communicating with its corresponding FMon at the PTC. The operations of a PE are time-driven after it has achieved full time synchronization with the BIU and the FTmr.

9.2.1. Main Functions

The PE provides three levels of services to the application: time synchronization, communication schedule update, and application-level communication. The time at a PE has two elements: the ROBUS Time (RT) given by the ROBUS time service, and the ROBUS Cycle Index (RCI) given by the FTmr. The RT is the current time within a ROBUS cycle, and the RCI is the number of the current cycle within a Control Cycle (CC). As described in Section 4.3.1.1, the protocol to synchronize the PE expands the ROBUS synchronization protocol (from which the PE gets the RT) by adding an exchange phase between the PEs and the FTmr to generate agreement among the PEs on the RCI.

Using the RCI and the CC-level communication schedule delivered by the PESU, the PEs can generate an RC-level communication schedule to program ROBUS during its Schedule Update minor mode. The non-faulty PEs will proceed with the actual communication of application-level messages only if they receive confirmation from ROBUS that the accepted schedule is equal to the desired schedule.

The purpose of the SUT application is to characterize the effects of ROBUS faults in the communication between active PEs. Section 4.3.1.2 provides a description of the application. Essentially, it consists of a three-step process. First, at the beginning of a control cycle, every active FMon (i.e., an FMon connected to an active PE) sends a TCM with application data to its corresponding PE. Section 7.2.2 describes the format of this TCM. Second, the PEs broadcast their data on ROBUS at their scheduled time in the control cycle. An application-level message sent by a PE consists of a header, the application data and a CRC-based frame check sequence (FCS). The header has three main fields: a sender field identifying the source PE, an error-detected-at-sender field set to 1 (i.e., true) if the sender PE detects an error in the message from its FMon, and an error-detected-at-receiver field set to 1 if the receiving PE detects an error in the message from the sending PE over ROBUS. The CRC for the PE message is computed by the sending PE itself over the whole message, including the header. Each PE message is broken down into individual words, and each word is transmitted on ROBUS as a single DATA-tagged ROBUS message (RM). In the final step of the application, for every scheduled source PE, a receiving PE recovers the application-level message, if possible, and forwards the received data, including an updated message header, to its corresponding FMon. It is assumed that each PE is scheduled to send at most one application-level message per control cycle. At the end of the control cycle, an active FMon outputs a set of observations for the communication paths from every PE to the FMon's PE. Section 10 provides additional information about the FMons.

9.2.2. Operational Modes

In the layered services structure of SPIDER, there is a hierarchical service dependency in which lower-level services must be operational before higher-level services can reach full operational status. The PEs can interact with one another at the application level only if ROBUS is operating in the Clique Preservation major mode (CPM) and the PEs have agreement on the current time. At a particular PE-BIU node, the service dependency chain requires that the BIU synchronize to a ROBUS clique, after which the PE can synchronize to the BIU at the ROBUS Time (RT) level. Next, the BIU must reach the CPM mode in order for the PE to be given access to the bus. Finally, the PE can then interact with other PEs to complete the agreement on the current time, as well as other PE-level state variables (e.g., the communication schedule). For the version of SPIDER currently implemented, which lacks PE-level distributed agreement protocols with communication over ROBUS, the FTmr at the PTC works as a central coordinator to generate agreement on the RCI.

Figure 9.3 shows the major mode transition graph for a PE. The Operational Coordination Level (OCL) is related to the degree of readiness of the PE to deliver application-level services. After a reset, the PE enters the base OCL0 level, where it remains until it has RT synchronization. In OCL1, the PE waits until the BIU has reached the CPM mode. OCL1 mode can be shorter than one ROBUS cycle if the BIU is participating in a full ROBUS initialization sequence. Otherwise, the BIU is executing a reintegration sequence, and the PE must wait several ROBUS cycles. The final state recovery step for the PE is to communicate with the FTmr to synchronize the RCI. From that point on, in OCL3, the PE is in a preservation mode to deliver user services indefinitely. At any time, the detection of processing error triggers an immediate cessation of processing until a PE-external action is taken (e.g., a state reset triggering a return to OCL0).

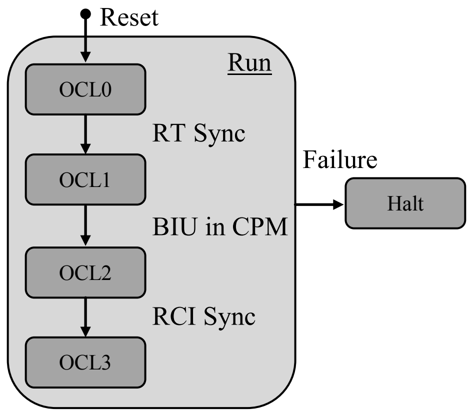


Figure 9.3: Major-mode transition diagram for the PE

Figure 9.4 shows the PE minor-mode transition graph. This graph captures the cycle of activities by a PE during state initialization and preservation according to the OCL major-mode flow. Starting from a reset, a PE must first synchronize its RT state variable using the sync message from the BIU. From that point on, the operation of the PE is ROBUS-time-triggered and tightly coupled to the operation of the BIU. Section 3.8 describes the activities at the interface between the PE and the BIU. After RT synchronization, the PE receives the ROBUS diagnosis results for the BIU and RMU nodes. The local BIU then reports its current operation major mode and its assigned Id. These two items of information allow the PE to track the BIU mode transitions and to know its own Id number (which is the same as the BIU's), respectively. If the PE has reached OCL3, it can properly communicate on the bus according to the configured schedule. Otherwise, it must simply preserve its RT synchronization. RCI synchronization for initialization or preservation is executed only if the PE is in OCL 2 or 3. The error checks and failure conditions monitored by the PE are a function of the OCL.

9.2.3. Organization and Error Checks

The internal organization of the PE in simplified form is illustrated in Figure 9.5. We describe the operation and interaction between modules for each minor mode.

After a reset, with the PE in OCL0, the Schedule Generator checks the content of the Control Cycle (CC) schedule buffer and circularly shifts the content of the buffer until it reaches the beginning of the schedule. An error in this check is a failure condition. If no error is detected, the Master Controller (MC) starts the Watchdog module and waits for a synchronization message from the BIU. If the synchronization message does not arrive before the Watchdog timer expires, a failure condition is triggered. Otherwise, the RT state variable is synchronized and the PE transitions to the OCL1 major mode. At this point, the PE restarts the Watchdog to expire after the maximum duration of one ROBUS cycle.

In the next minor mode, the MC receives the ROBUS diagnoses for the BIU and RMU nodes. The MC checks the content of these messages and triggers a failure condition if the acceptance criteria are not satisfied.

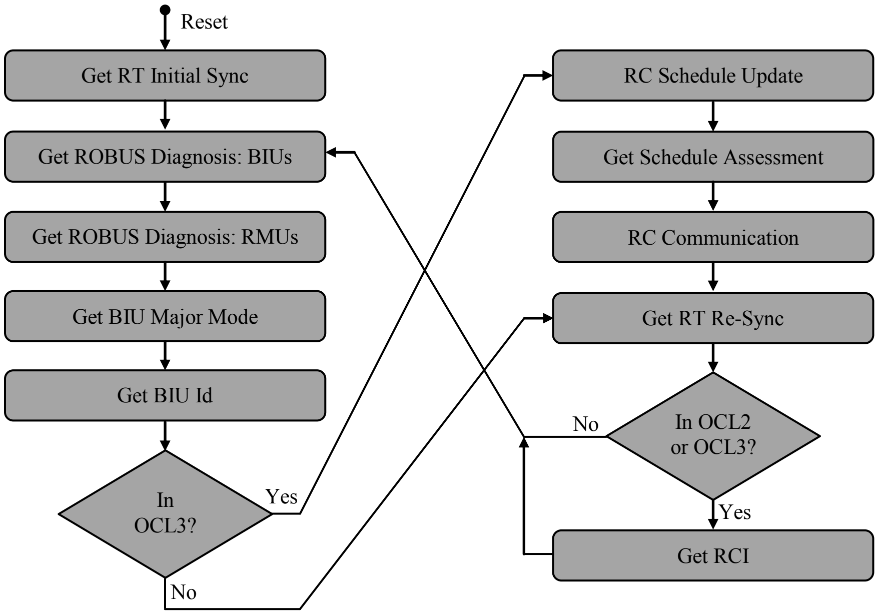


Figure 9.4: Minor-mode transition diagram for the PE

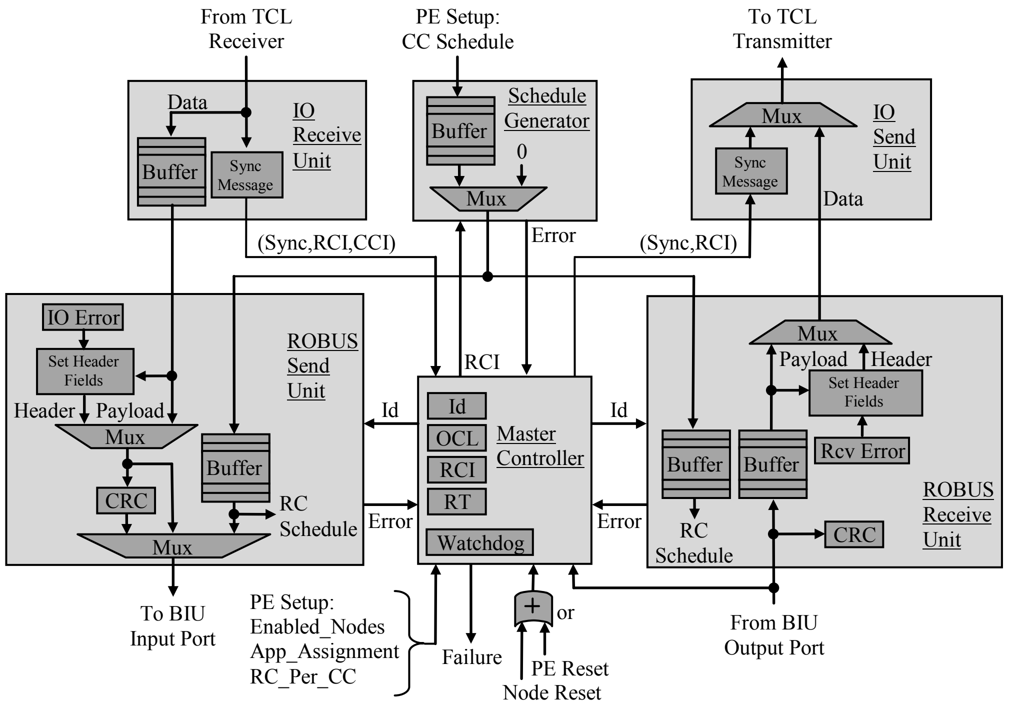


Figure 9.5: Block diagram for Processing Element

The mode messages from the BIU are always checked for valid content and any violation is treated as a failure condition. Once the PE has reached OCL1, it allows at most two ROBUS cycles for the BIU to reach the CPM mode. If this condition is not satisfied, the PE declares a failure. In OCL3, the PE only accepts CPM mode messages from the BIU, and a failure condition is asserted if the BIU reports any other mode.

The first time the Id message is received by the PE, it is checked for proper content. The Id value is valid only if it is between 1 and the number of PEs in the system. From the second time that the PE receives the Id message, the Id value must match the one received the first time. A violation of these acceptance checks triggers a PE failure condition. If the Id is accepted the first time it is received, the value is used to determine whether or not the PE is to remain active based on the entries in the Enabled_Nodes configuration parameter received from the PESU. If the PE is not to remain active, it ceases all activity and goes to an idle state. Otherwise, it continues with the next minor mode.

If the PE is not in OCL3, it is not ready to communicate with other PEs. In that case, it does not actively participate in updating the ROBUS communication schedule or in sending messages on ROBUS. In those minor modes, the PE simply monitors the RMs from the BIU for validity relative to the protocols used by ROBUS.

To re-synchronize the RT state variable, the PE waits for a synchronization message from the BIU and performs a content validity check before applying a correction to the RT variable. If a received message is rejected or the watchdog timer expires before the sync message arrives and the local RT is updated, the PE declares a failure and halts its operation. If the PE is in OCL2 or OCL3, it will send a TCM sync message to the FTmr using the IO Send Unit. The TCM contains the value of the RCI state variable at the MC. The PE will then wait to receive a sync message from the FTmr containing the current value of the RCI as well as the Control Cycle Index (CCI), which is not used by the SUT in the current implementation. A failure condition is declared if the received sync message does not satisfy the acceptance criteria, including a sync message with RCI set to 0 within RC_Per_CC ROBUS cycles from the time the PE entered the OCL2 mode. In OCL3, the arrival of a valid sync message with a new RCI triggers the generation of the next RC communication schedule by the Schedule Generator. The ROBUS Send and Receive Units load this schedule onto their own separate buffers and wait for a signal from the MC to begin processing in the Schedule Update mode.

In the Schedule Update mode, the BIU reads the schedule from the ROBUS Send Unit (RSU) according to a time-triggered execution pattern. Using the Id from the MC, the RSU records the number of messages in the schedule for this PE to be used later on in the PE data communication mode. The MC and the ROBUS Receive Unit (RRU) perform acceptance checks on the content of the scheduling results received from the BIU. The RRU checks include a one-for-one comparison of the received schedule against the submitted schedule. Any discrepancy in this check triggers an error signal, to which the MC reacts by declaring a failure and stopping the processing. While checking the schedule update results from ROBUS, the RRU also reduces the RC Schedule buffer content by eliminating any 0 entries. The remaining entries are the actual schedule to be executed during the communication mode (see Section 3.7.3).

The schedule assessment reported by the BIU is inspected by the MC to confirm the validity of the content and that the assessment corresponds to the schedule submitted (i.e., “zero” if all the entries in the schedule are 0, but otherwise an assessment of “valid”). An unexpected schedule assessment message from the BIU, in content or timing, triggers a failure condition and a halt of PE execution.

During PE communication in a ROBUS cycle, the PEs access the bus one at a time according to the agreed upon schedule and all the PEs receive all the scheduled messages. The IO Receive Unit (IORU) buffers the PE-message section in the payload of the TCM data message from the FMon (see Section

7.2.2). Under fault-free conditions, the message transmitted by the PE on ROBUS is the content of the IORU buffer, which includes the message header and payload, followed by a CRC word. The RSU replaces the content of the sender Id field with the Id from the MC, it sets the Error-Detected-at-Sender-PE according to its error check results, and it clears the Error-Detected-at-Receiver-PE field. If the IORU does not receive a valid data TCM from the FMon before the scheduled time for transmission on ROBUS, the IORU data buffer will be empty or incomplete and the RSU uses whatever value happens to be at the output of the buffer as the message payload content. At the scheduled transmission time, the BIU begins reading the message data from the RSU prompting for a new word as the previous one is broadcast on ROBUS as a DATA-tagged RM. For each PE, the number of scheduled RMs is equal to the size of the application-level message to be transmitted by the PE, including the header, payload and CRC.

At the receiving end, the MC checks the validity of the individual received RMs, while simultaneously the RRU uses the schedule to recover the application-level messages in the stream of received RMs. An RM error detected by the MC is a failure condition due to a fault in ROBUS and/or the BIU and triggers an execution halt. The RRU flags an application-level message as incorrect if any of the corresponding received RMs has a tag other than DATA or the CRC computed over the data section of the RMs (i.e., not including the tag field) does not match the received CRC in the last RM of the application-level message. The RRU buffers the received data words as they arrive, except for the CRC, and then records the PE source Id and error check result before requesting a TCM data message transmission by the IO Send Unit (IOSU). When the IOSU is ready, the RRU overwrites the sender Id and Error-Detected-at-Receiver-PE fields and forwards the content of the message to the IOSU, which sends it over to the FMon.

9.3. Embedded Node Monitor

The function of the ENM is to collect state information from the node and send it to the main State Monitor (SMon) at the PTC. The ENM is a key system component used for debugging during system development and for event analysis during experiments. The ENM must be easily reconfigured at design or synthesis time to change the data to be collected and the trigger conditions for sending the data to the SMon. The design and setup of the ENM must also minimize any possible interference with the normal operation of the SUT components, especially the PE, as well as the fault injectors.

Figure 9.6 is a simplified block diagram for the ENM at a PE-BIU node. The central element of the ENM is the Parallel-Input Serial-Output (PISO) buffer where the payload words for the TCM, including the TCM header and the state data (see Section 7.4), are loaded in parallel and shifted out as the message is read by the PMCU transmitter of the PTL. The sequence number is an 8-bit count that is incremented every time a new message is generated. The data registers are independently loaded with state information for the next message. This allows data from different sources (e.g., RPP, PE, etc) to be recorded at different times based on context-dependent triggers. This two-stage data buffering with independent data registers and a PISO buffer also enables the collection of new state data while a message is in the process of being transmitted. When a message trigger occurs, any data register that does not already contain updated state data will be automatically loaded with the current state at that time independently of the data source context.

In the current implementation, the ENM uses a priority encoder to rank the set of defined message transmission trigger events. The trigger events, in order of priority from highest to lowest, are the following: failure of the RPP, failure of the PE, and RPP mode transitions to Self-Test Mode (STM), Clique Detection Mode (CDM), Clique Initialization Mode (CIM), and Schedule Update in Clique Join Mode (CJM) or Clique Preservation Mode (CPM). If multiple triggers occur simultaneously, only the highest priority one will be reported in the TCM sent to the SMon.

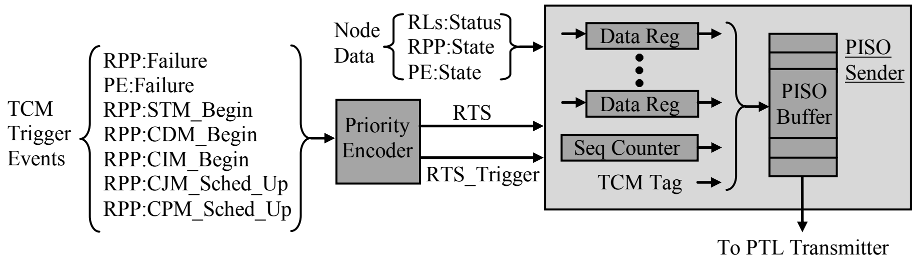


Figure 9.6: Block diagram for the Embedded Node Monitor

10. RMU Node

The main purpose of an RMU node in the SPIDER-based SUT is to perform the ROBUS communication function of an RMU component. In addition, an RMU node includes functionality to support the CFIMS processes of configuration, fault injection and state monitoring. As a ROBUS component, an RMU does not perform application-level functions. An RMU node must support the CFIMS setup phase, in particular the SPIDER Initialization and Fault Injection Setup modes (see Section 5). For the current implementation on RSPP1, the RMU node's functionality is implemented as hardware processes running on an FPGA.

Figure 10.1 shows a block diagram for an RMU node. The components are divided into the SUT Functional Node group and the CFIMS Embedded Modules group. The RPP configured at design-time as an RMU is the central component of the functional node. The RPP is supported by a set of WMCUs (one per RL) to communicate with the BIUs and a Reset module that it triggered by an asserted failure condition signal from the RPP. The output of the Reset module is routed through Fault Injector 0 to enable the simulation of fail-stop behavior. Other fault injectors may be added to simulate more complex failure patterns. The ENM collects data from the local components and sends it to the State Monitor at the STC for debugging and event analysis. The PMCU is shared by local components. Currently, the PMCU is configured with two transmission ports with the ENM connected to the highest priority port and the other port disconnected and available for future use.

In terms of components, an RMU node has a subset of the components in a PE-BIU node. The RPP, fault injectors and ENM in an RMU node are the same as in a PE-BIU node. Section 9 and other sections of this report provide further details about the design of the RMU node components.

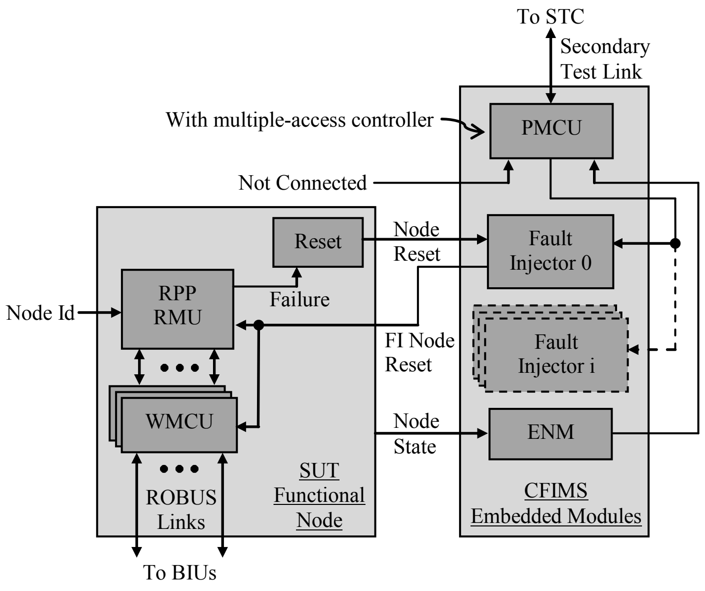


Figure 10.1: Block diagram for RMU node

11. Hardware Processes of the Primary Test Controller

The CFIMS must provide four main functions: system (i.e., SUT and CFIMS) reconfiguration, SUT function testing, SUT fault injection and SUT monitoring. In the current version of the CFIMS, these functions are realized by two test controllers (Primary and Secondary) and embedded processes at the SUT nodes. With the available and planned RSPP execution platforms, multiple test controllers are necessary to support SUT configurations with a number of SUT nodes larger than the number of custom communication ports in one RSPP node (i.e., eight). As described in Section 4.3, at the functional level, the PTC and STC consist of a set of intercommunicating hardware and software processes with the hardware processes performing lower-level and timing-sensitive operations and the software processes performing function management activities with more relaxed execution timing constraints. This section covers the design of the PTC hardware processes. Section 12 covers the design of the STC hardware processes. The PTC and STC software processes are described in Section 13.

The PTC hardware processes run on the FPGA of the RSPP node configured as the PTC. These processes have three main interfaces to communicate with other processes: the RSPP node's peripheral bus (i.e., ISA in RSPP1 or PCI in RSPP2) for the PTC software processes, the CCL for the STC, and the PTLs for the PE-BIU nodes.

The PTC software processes interact with the hardware processes to configure the hardware according to the content of the Test Specification file, supply fault-injection test vectors during execution, collect execution data for post-test analysis and real-time monitoring by the test operators, and start and stop a round as determined by the test operator (see Section 13). The interface between the software and hardware processes must take into account the difference in execution rate and response delays between the hardware and software processes to ensure efficiency of interaction without loss of data.

The CCL is used to coordinate the operation of the PTC and STC, including system configuration, fault injection and round control. The overall objective in this coordination is to allow the controllers to behave as a single large controller from the viewpoint of the SUT. The Controller Coordination Protocol (CCP) described in Section 5 is used for overall coordination of round execution for the hardware and software at the PTC and STC. Once the software enables the hardware controllers, they assume command of round execution using the CCP to coordinate their actions, while the PTC and STC software processes become closely coordinated indirectly as client (or slave) processes tracking the execution of their respective hardware processes. Besides the round control processes, the CCL is also used by the function testing and fault injection processes to achieve coordinated interaction with all the SUT nodes. Thus, the CCL is a shared communication resource requiring mechanisms to ensure minimum access interference between processes and complete and correct transactions every time the link is accessed.

The PTLs are the direct communication links between the PTC and the PE-BIU nodes. These links are used for PE-BIU node setup, function testing, fault injection and state monitoring. As is the case for the CCL, every PTL is a shared communication resource with the need for access coordination mechanisms. The message traffic on the PTLs differs from that on the CCL mainly in the absence of the high-priority CCP messages and the increased traffic volume due to the addition of PE synchronization and application-level messages in both directions at a rate determined by the duration of the ROBUS cycle and the control cycle. The selected TCL access-coordination mechanism must satisfy the added requirement of not having a significant impact on the complexity of system design or analysis. The number of PTLs is equal to the number of PE-BIU nodes specified at synthesis time, generally denoted by N. Given the current CFIMS design and the number of custom communication ports in a RSPP node, the maximum number of PE-BIU nodes that can be supported is seven (with the eighth communication port used for the CCL). SUT configurations with a larger number of nodes require a redesign of the test controllers.

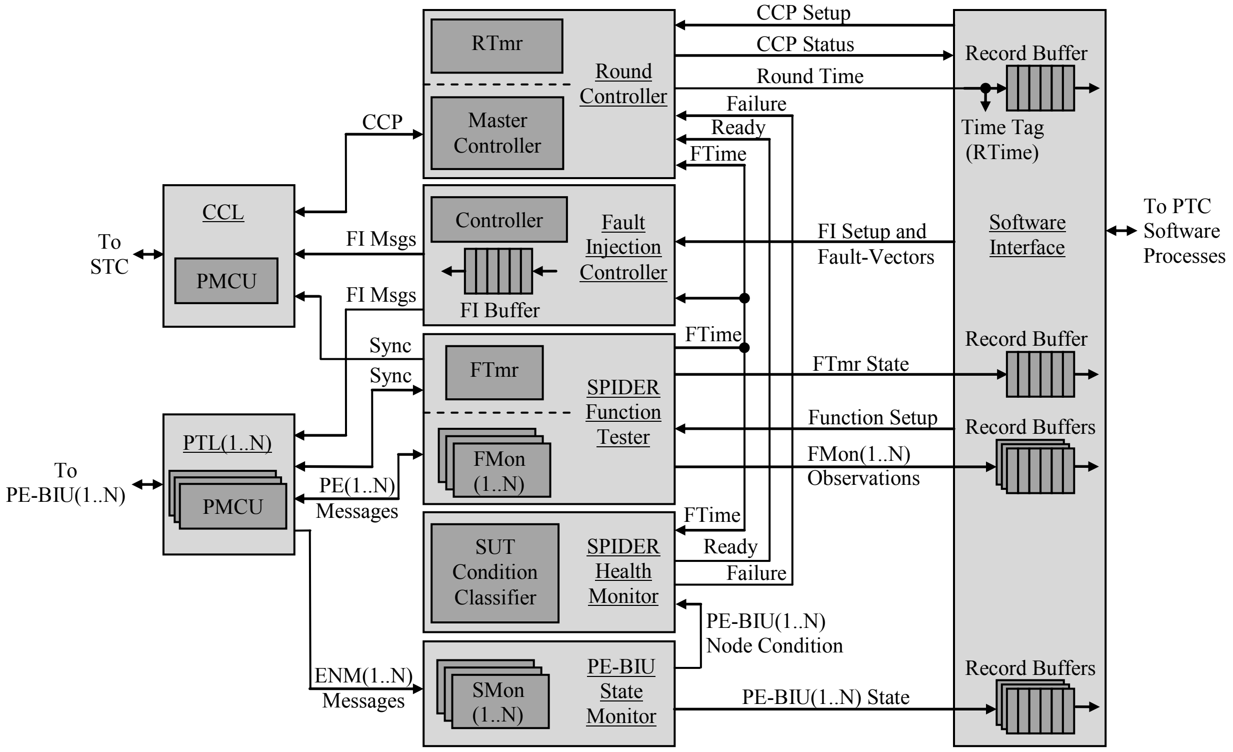


Figure 11.1: Block diagram for the Hardware Processes of the Primary Test Controller

Figure 11.1 shows an abstract high-level view of the PTC hardware processes. The external interfaces include the CCL and PTLs to communicate with remote hardware processes (i.e., running on different physical nodes), as well as the interface to the local software processes. The Round Controller module provides overall round-level coordination for the local hardware and software processes, and uses the CCP to coordinate its actions with the remote Round Controller at the STC. The Round Controller also provides a global time-reference service in the form of a Round Timer (RTmr) used to time tag observations. The STC contains a similar RTmr, and both are continually synchronized throughout a round via the CCL. The software collects data with snapshots of the Round Time (RTime) state from which it is possible to reconstruct the round timeline during post-test analysis. The SPIDER Health Monitor (HMon) supports the Round Controller as a dedicated monitor to check the condition of the SUT. Specifically, the HMon confirms the successful initialization of the SUT during the system setup phase in a round and, thereafter, it continuously monitors the SUT to detect unexpected responses that are indicative of a physical or functional failure that requires a halt to the execution. The PE-BIU State Monitor (SMon) receives state data sent by the ENMs and forwards it to the software. Each SMon lane also performs a condition assessment on its corresponding PE-BIU node based on the received state records to determine whether the node is operating normally or undertaking a recovery process. The SPIDER Function Tester (SFT) contains the Function Timer (FTmr), which serves to synchronize the PEs and provide an SUT-based time reference (called the FTime) to the test controllers, and the Function Monitors (FMons), which interact with the PEs at the application level. The FTmr and FMons generate data records for collection by the software. The Fault Injection Controller (FIC) configures the fault injectors at the SUT nodes (i.e., PE-BIU and RMU) and controls the timing of injection based on the fault specifications obtained from the software. The configuration and injection specifications from the software are buffered locally at the FIC to accommodate the difference in timing response between the hardware and software. The PMCUs for the TCLs (i.e., CCL and PTLs) use multiple-access controllers to arbitrate access to the links. The PTC functional modules are also programmed to minimize access interference (i.e., collisions) by using time-triggered operation (relative to the FTime) when possible. However, the modules must also be capable of handling event-triggered transmit and receive transactions (e.g., fault injection or state record arrival) to deal with unpredictable SUT behavior due to injected faults.

The following subsections expand on the design of the PTC components.

11.1. PTC Round Controller

The purpose of the Round Controller (RCtlr) is to provide overall round-level coordination for the CFIMS test controllers. The PTC RCtlr interacts with and monitors the local hardware processes, the test software, and the STC RCtlr throughout a round to maintain coordinated action and ensure proper completion of the round.

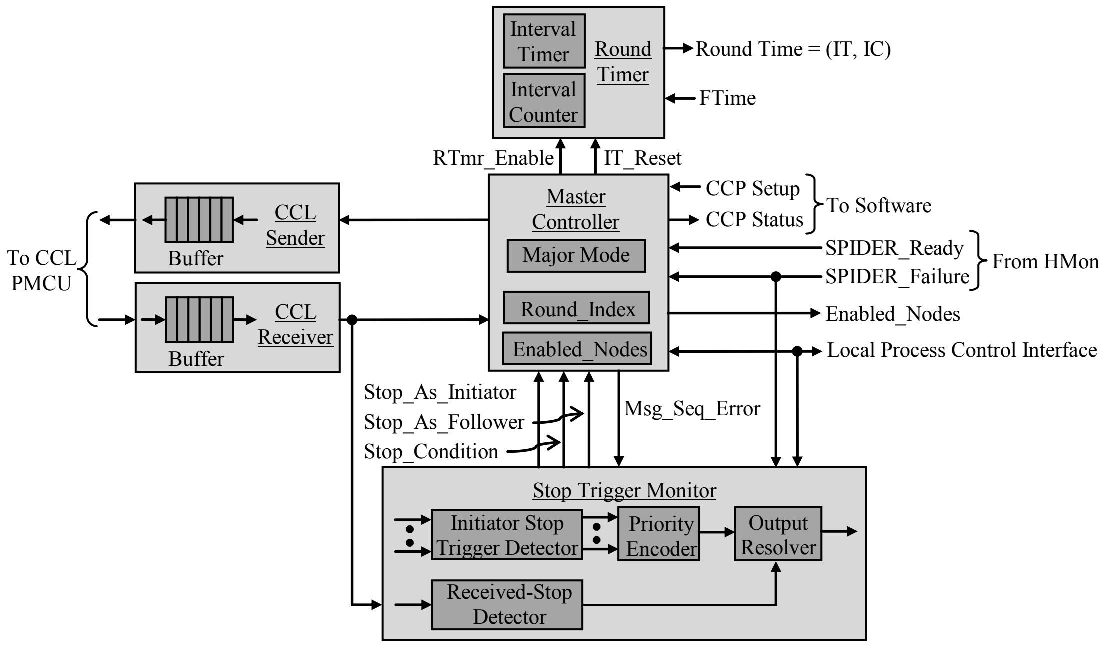


Figure 11.2: Block diagram for the PTC Round Controller

Figure 11.2 shows a block diagram for the PTC RCtlr. The Master Controller (MC) is the central execution coordinator. The MC's major mode transition graph is shown in Figure 11.3. The software loads the Enabled_Nodes (which specifies the SUT nodes to be active in the round) and Round_Index variables before enabling the controller to begin the round. From that point forth, the MC outputs the CCP status to allow the software to follow the progress in the execution of the round. To interact with the CCL, which is shared with other PTC hardware processes, the RCtlr uses dedicated send and receive modules to relieve the MC from the tasks of waiting for CCL transmitter access arbitration and received-message filtering. The CCL receiver loads only CCP messages, which have TCM tag fields of RC_ENABLE, RC_READY, RC_START, and RC_STOP. The HMon asserts the PE_BIU_Ready signal when SPIDER is initialized. The SPIDER_Failure signal can be asserted at any time after that point if any of the monitored failure criteria at the HMon is satisfied. The Stop Trigger Monitor at the RCtlr constantly monitors for any of a defined set of round stop conditions. The defined stop conditions include the normal completion of a round as indicated by the FIC (when all fault test vectors have been applied) or the SFT (when all specified control cycles have been completed), a CCP message sequence error reported by the MC, a local hardware or software process stop request (on a detected error or some other condition), SUT failure as detected by the HMon, and a received RC_STOP TCM from the STC while the local MC is in normal System Run operational mode. Appendix B lists all the stop conditions defined in the current system implementation. If the stop is triggered by a local event, the RCtlr begins the round stop sequence as the initiator and sends an RC_STOP message to the STC to stop the execution there too. If the local stop is triggered by a received RC_STOP message, the initial stop trigger was detected at the other Test Controller and the local RCtlr executes the round stop sequence as the follower. In either case, the Stop Trigger Monitor reports the highest priority stop triggering condition, and this is then relayed to the software and the remote node (in the case of stopping as the initiator). The CCP does not provide a mechanism to reach agreement on the stop condition if the PTC and STC simultaneously stop as initiators. In that case, the system will stop, but the reported stop condition may be different at the PTC and STC. Section 5 gives a detailed description of the CCP.

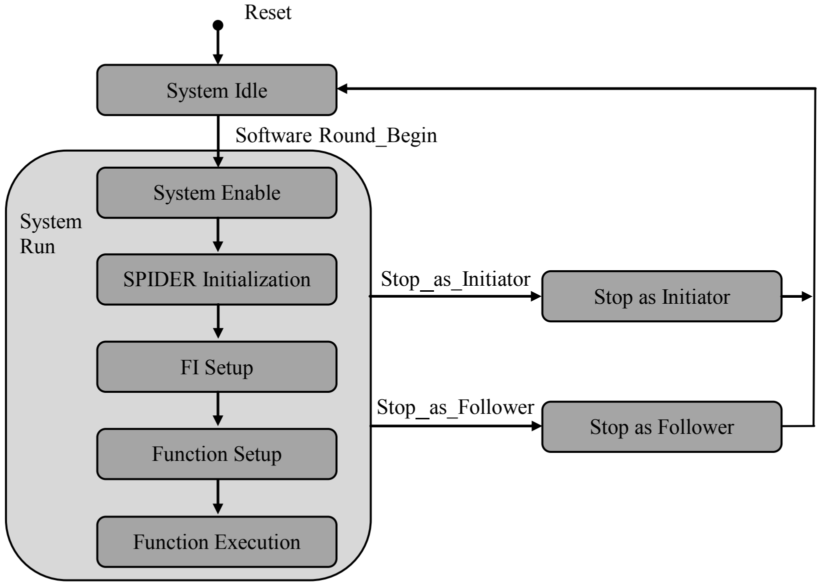


Figure 11.3: Major mode transition graph for the Master Controller of the PTC RCtlr

The Round Timer (RTmr) generates the global time reference used to time tag observations. The RTmr measures the time relative to particular reference events. The Round Time (RTime) has two components: the Interval Time (IT) that measures the time since the last reference event, and the Interval Count (IC) that counts the number of times the IT has been reset upon a reference event. The MC enables the RTmr in the System Enable major mode. That is the first reference event. During the setup phase, the IT is reset when each of RC_ENABLE, RC_READY and RC_START is transmitted. Once the round enters the execution phase, the RTmr is reset a short delay after the FTmr sends out the Sync message to the STC at the beginning of every ROBUS cycle when the FTmr is in Preservation mode. This delay compensates for the message propagation delay over the CCL and is intended to synchronize the RTmrs at the PTC and STC. The RTmr is stopped only at the end of the round.

11.2. SPIDER Function Tester

The SFT interacts with the PEs to coordinate their actions, provide the application workload and monitor their operation at the application level. The SFT is responsible for configuring the PEs during the round setup phase and then enabling them at the beginning of the execution phase. The SFT must also be able to interact with the round controller to implement the CCP, with the software for setup and data collection, and with the other CFIMS hardware processes to provide the FTime used as a time reference for interacting with the SUT.

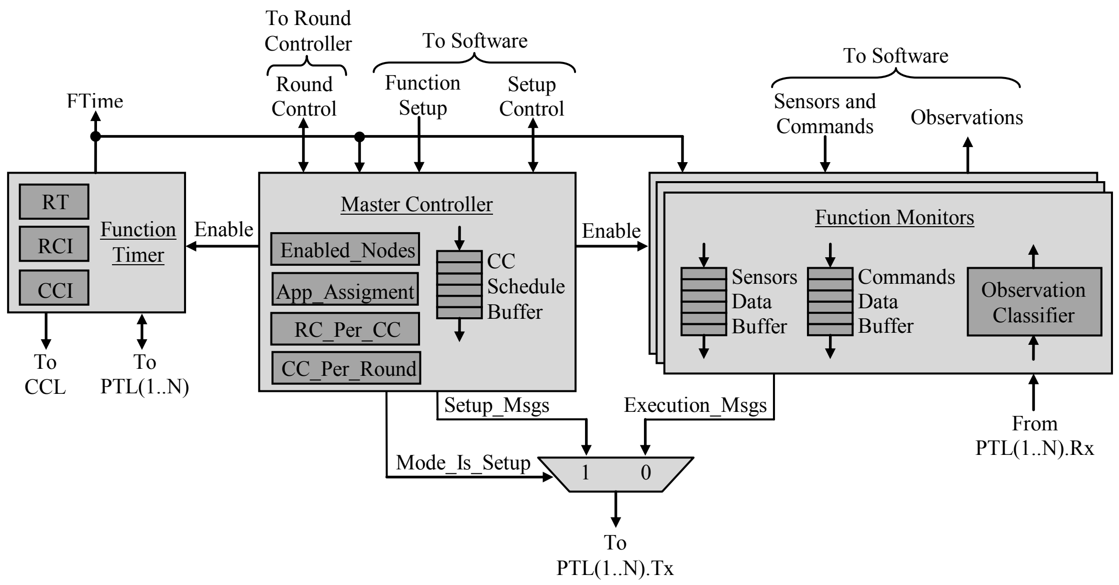


Figure 11.4: Top-level block diagram for SPIDER Function Tester

Figure 11.4 shows a block diagram for the SFT. The Master Controller (MC) handles the interaction with the Round Controller, the setup of the PEs and the enabling of the FTmr and FMons specialized modules. The Enabled_Nodes vector is received from the Round Controller at the beginning of the round. During the Function Setup mode, the software loads the function configuration data from the Test Specification, including the Application_Assignment, the number of ROBUS cycles per control cycle RC_Per_CC, the round duration CC_Per_Round, and the PE communication schedule for a control cycle. The software also loads on each FMon the sensor and command data to be used during the execution phase. When the software is finished loading the function parameters, the MC builds the SF_SETUP TCM and sends it to the PEs. At the beginning of the Function Execution mode, the MC enables the FTmr and the FMons, and sends the SF_EXECUTE TCM to enable the PE. From then on, the MC waits for the FTmr to complete the round duration specified by the software, or for a round stop command from the Round Controller. A CC_Per_Round value of 0 means that the FTime will not be used to limit the duration of the round and that the round stop trigger will occur elsewhere. The FTime consists of three elements: the ROBUS Time (RT), the ROBUS Cycle Index (RCI) and the Control Cycle Index (CCI). When the FMons are enabled, they use the FTime as a reference to trigger major transitions in their internal operation. An FMon sends a sensor or command data message to its corresponding PE (depending on the PE's application assignment) at the beginning of the control cycle, and then opens a reception window to listen for and classify the observations for received PE data messages. When the RCtlr issues a stop command, the SFT MC requests the FMons to stop.

11.2.1. Function Timer

The primary purpose of the FTmr is to provide the CFIMS with an SUT-referenced time (the FTime) service that can be used to coordinate distributed actions. Given that the PEs in the current SUT implementation do not execute PE-level coordination protocols with communication over ROBUS, the FTmr function is expanded to also provide a distributed coordination reference service to the PEs.

The FTime is intended to measure the SUT function execution time from the moment the SUT is enabled by the CFIMS in the Function Execution mode. The FTime differs from the RTime in that the FTime measures the total time during which the SUT has been making forward progress in executing the application without including execution discontinuities when the system as a whole is recovering from injected faults (and so, is not executing the application), while the RTime measures the total elapsed real time since the round was enabled. The structure of the FTime is related to the levels of services in the SUT. The FTime is defined to consist of three elements: the ROBUS Time (RT) measuring the elapsed time since the latest distributed synchronization event generated by the ROBUS synchronization protocol; the ROBUS Cycle Index (RCI) measuring (or counting) the number of ROBUS cycles completed since the beginning of the "current" application-level control cycle; and the Control Cycle Index measuring (or counting) the number of control cycles since the SUT function execution was enabled.

The FTmr gets the RT from the PEs using the expanded version of the ROBUS synchronization protocol described in Section 4.3.1.1. The FTmr must have the capability of initializing its time when the CFIMS enters the Function Execution mode. Also, because of the possibility of ROBUS (and the SUT as a whole) losing internal coordination among its nodes due to injected faults and then entering a system recovery mode, the FTmr is designed with the capability to reacquire RT synchronization after a temporary disruption of SYNC_PE_TIME TCMs from the PEs. Figure 11.5 shows the mode transitions for the FTmr. The Initialization mode applies to recovery starting from reset and re-initialization from a detected failure. This mode is functionally similar to the Clique Detection Mode of ROBUS-2 (see [Torres05A]) and is concerned mainly with getting the RT from the PEs. Once RT synchronization has been established, the FTmr transitions to Preservation mode to maintain synchronization precision by means of periodic re-synchronizations. Additional details about the implementation of the synchronization protocol at the FTmr and the failure conditions that trigger transitions to the Initialization mode are given later in this section.

The management policy for the RCI and CCI at the FTmr must cover the operational disruptions caused by detected failures. Figure 11.6 illustrates the chosen rules for managing these FTime elements. Notice that RCI and CCI do not increment in the Initialization mode (even when there is an initialization failure), and CCI is incremented when there is a transition from Preservation to Initialization mode. From the perspective of the FTime, this means that a control cycle is valid and complete the moment it is started irrespective of its actual duration, which is determined by events happening during the cycle. This enables for even a partially executed control cycle to have a unique CCI value.

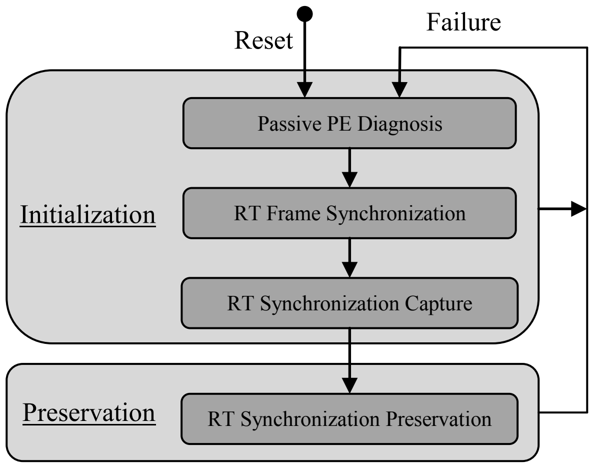


Figure 11.5: Mode transition graph for the Function Timer

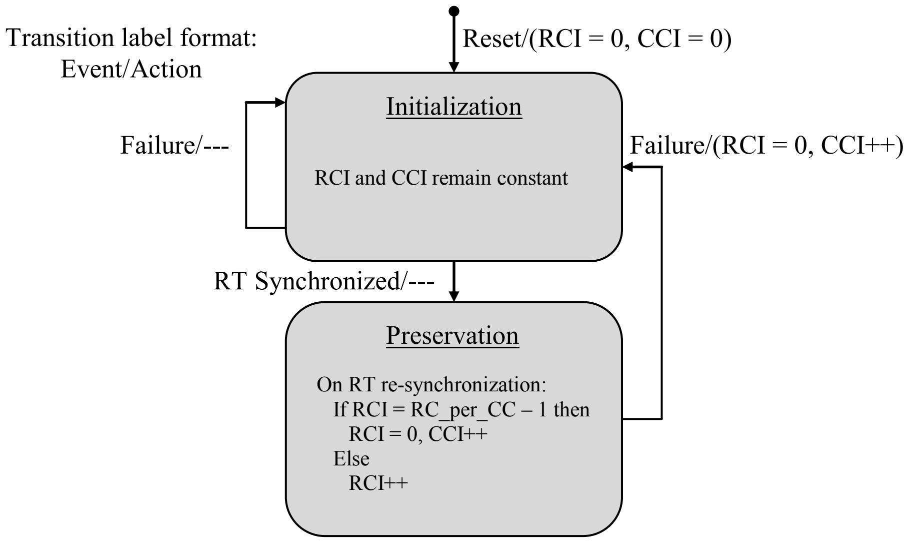


Figure 11.6: Policy rules for managing the values of RCI and CCI

The RT synchronization process at the FTmr is based on the use of a middle-value-select event voter with a dynamic eligible voter set. The voted events are the times of arrival of the SYNC_PE_TIME TCMs from the PEs. The eligible voters are the subset of PEs whose inputs are considered in performing the vote. Voting eligibility of individual PEs is determined from the diagnosis of message traffic observations at the FTmr. To maximize the tolerance to faulty inputs, the diagnostic subsystem must satisfy the property of correctness, which states that a PE shall be diagnosed as an ineligible voter only if it is physically faulty or its state is incorrect. This ensures that fully operational PEs are not removed from the eligible voter set. However, under conditions of less than perfect integrity, the FTmr does not achieve 100% error detection and diagnosis coverage and some bad PEs may remain in the eligible voter set. The fundamental requirement of the event voter to guarantee output validity is that a majority of eligible voters be trustworthy (i.e., can be relied upon to provide correct inputs from which to correctly compute and deliver the expected service). If it is not true that the majority of eligible voters are trustworthy, there is no guarantee of correct voter output as the fault-masking ability of the voter (i.e., the ability to deliver a valid output even in the presence of invalid inputs) is compromised. In that case, for the current FTmr design, it is assumed that the voter output is unreliable and a failure has occurred. Furthermore, in accordance with the model of a fault-causing phenomenon of known bounded duration used in the design of ROBUS-2 (see [Torres05A]), it is assumed that the failure condition is not persistent (i.e., it is transient) and can be corrected by a re-initialization of the system.

The purpose of the FTmr diagnostic capability is to identify untrustworthy PEs as soon as possible so they can be removed from the eligible voter set to preserve the majority-trustworthy condition. The diagnostic capability is also required to detect when such condition is not true and then trigger a re-initialization. Note that the CFIMS fault injection targets are only the SUT nodes. The FTmr itself is never directly affected by faults. This implies that the diagnosis performed by the FTmr is really a diagnosis of the PEs and the SUT as a whole, and that when the FTmr declares a failure, what has actually failed is the SUT. This is why the SPIDER Health Monitor (see Figure 11.1 and subsection 11.5) uses the FTime to check the SUT health status in the Function Execution mode of the round.

The FTmr has two modes of operation relative to the ROBUS Time (RT): unsynchronized and synchronized. Unsynchronized operation happens only in the Initialization mode. In the Passive PE Diagnosis minor mode, the FTmr opens two consecutive observation windows, each with duration equal to the longest ROBUS cycle. For each input PTL lane, the FTmr expects to receive one or two SYNC_PE_TIME TCMs in each observation window. If this is not the case, the corresponding PE may be accused of being bad (i.e., exhibiting incorrect operation). In the RT Frame Synchronization minor mode, the FTmr searches for the gap between clusters of SYNC_PE_TIME TCMs from trusted PEs. The trustworthy PEs send one cluster of synchronization TCMs every ROBUS cycle. The FTmr shall find the gap between consecutive message clusters in less than one ROBUS cycle. While the FTmr is actively searching for this gap, it expects to receive at most one synchronization message from each PE. The arrival of two or more messages from a PE is sufficient evidence of incorrect operation by that PE. In the RT Synchronization Capture minor mode, the middle-value-select event voter is enabled with the initial set of input eligible voters (IIEVs) including those PEs that have not been accused up to that point. The voter is designed to operate under the assumption that a majority of the trusted input eligible voters (IEVs) are indeed trustworthy. If that is so, a majority of the received Sync events shall fall within a known bounded time distance from the selected middle event. This bound on the synchronization precision for trustworthy PEs is computed by analysis prior to system synthesis. A violation of this synchronization precision property detected at the time the event voter generates an output is a failure condition that triggers a restart of the Initialization mode. In addition, any one sync event that is not within a particular bounded time distance of the middle event (also determined by analysis prior to system synthesis) is sufficient evidence to accuse the corresponding PE of incorrect operation. If it is determined by these accusations that less than a majority of IIEVs were trustworthy, that is also a failure condition that triggers a re-entry into the Initialization mode. Finally, given the periodicity of execution of the ROBUS Synchronization Preservation protocol, it is expected that the event voter will generate an output within one ROBUS cycle from the time it is enabled. Otherwise, there is sufficient evidence that the IIEV was invalid, and a re-initialization shall be started. In this unsynchronized mode, the FTmr behaves similarly to the ROBUS Protocol Processor (RPP). Additional information can be found in [Torres05A] and [Torres05B].

In synchronized operation in the Preservation mode, the FTmr expects to receive synchronization messages from the PEs during a small time interval determined based on the ROBUS cycle period and the achieved RT synchronization precision between the PEs and the FTmr. For each PTL lane, if a synchronization message arrives outside of this expected reception window, or anything other than one synchronization message arrives during the window, the corresponding PE is accused of incorrect behavior. After the window closes, the event voter is enabled to operate with the latest set of trusted inputs, and the output is used to synchronize the RT at the FTmr. If at any time in the Initialization or Preservation mode the FTmr trusts none of the PEs, it declares a failure and returns to the Initialization mode.

Figure 11.7 shows a block diagram for the Function Timer. The FTmr receives SYNC_PE_TIME TCMs from the PEs arriving at the PTC over the PTLs. These TCMs contain two items of information: the RCI at the sending PE, and the time of arrival of the message, from which the FTmr can infer the time at which the PE sent the message using known bounds on the PTL communication delay. A TCM Receiver module rejects any TCM with a different tag or errors indicated by the PTL PMCU receiver. When a TCM Receiver receives a good TCM, the RCI is loaded onto a register and a one-tick "Sync" pulse is generated a fixed delay after the message is received. Only the pulse is forwarded to the rest of the FTmr.

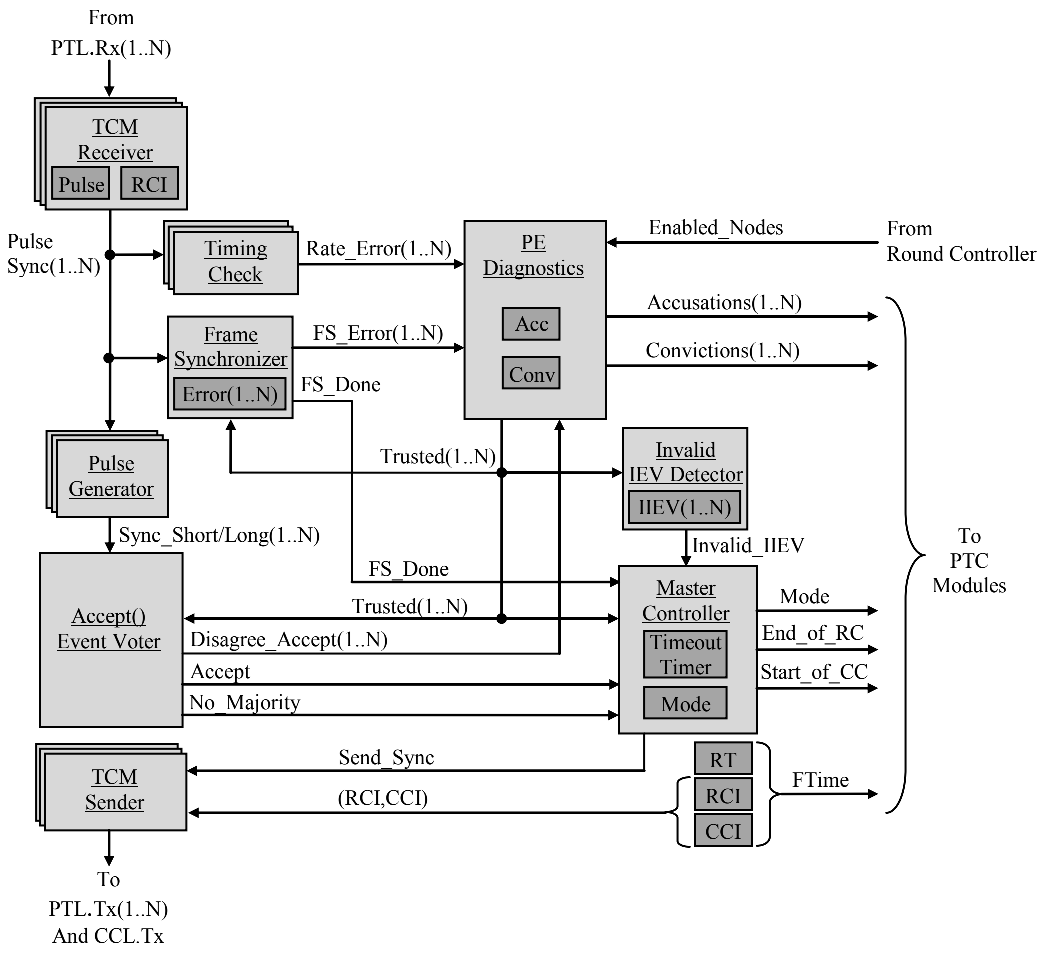


Figure 11.7: Block diagram for the Function Timer

The Timing Check blocks are responsible for counting the number of received PE synchronization messages during unsynchronized and synchronized reception windows. When a window is closed, this module compares the actual number of received messages against the allowed range and signals an error for each lane with a number of inputs outside the valid range.

The Frame Synchronizer block reads the SYNC_PE_TIME TCMs from PEs and executes the frame synchronization protocol described in Section 7.2.1 of [Torres05A]. When the frame gap is found, an error report is generated for each input channel and the Master Controller is informed (by means of signal FS_Done) to transition to the RT Synchronization Capture minor mode.

The PE Diagnostics module applies a diagnostic policy similar to the one used in ROBUS-2 (see [Torres05A], Section 4). A PE is trusted if it is enabled, not accused and not convicted. The Enabled_Nodes vector indicates which PE-BIU nodes are active. A PE is accused when an error is detected in its behavior. In the Initialization mode, all the active PEs are initially trusted. If an error is detected for a particular PE, that PE is accused and the accusation remains effective until there is a transition from the Initialization mode (including a transition back to the same mode). Once a PE is excluded from the trusted set in the Initialization mode, it remains excluded for the duration of that mode. There are no convictions in the Initialization mode. An accusation in the Preservation mode is effective from the time the error is reported to the PE Diagnostics module until the end of the ROBUS cycle at the completion of execution of the RT synchronization protocol. In the Preservation mode, an accusation is promoted to a conviction starting at the end of the ROBUS cycle in which the accusation occurred and lasts for the duration of one ROBUS cycle. As long as an enabled PE continues to be accused, it will be excluded from the trusted set. Readmission is possible only after the PE completes one ROBUS cycle with no new accusations. When that happens, the conviction will be cleared (because there are no new accusations to sustain it) and trust will be re-established. The Accusations and Convictions vectors are available at the outputs of the FTmr as part of the records collected by the PTC Software Interface for post-test analysis.

The Invalid IEV Detector module compares the trusted sets at the beginning and at the end of the RT Synchronization Capture minor mode to determine if there was a violation of the trustworthy-majority assumption for the eligible voters used by the event voter. A detected violation triggers a re-initialization of the FTmr.

The Master Controller (MC) is the central coordinator of operation within the FTmr. The MC controls the FTime elements RT, RCI and CCI. At the appropriate time, the MC commands the TCM Senders to send a SYNC_ROUND_TIME TCM containing the RCI and CCI. The state information output by the MC is used by PTC modules reading the FTime and for data records collected by the PTC Software Interface.

The Pulse Generators and the Accept() Event Voter require special consideration in their description as their operation and implementation are meant to be size and time efficient, but are not entirely intuitive. When the FTmr is expecting to receive synchronization messages from the PEs in the RT Synchronization Capture and RT Synchronization Preservation minor modes, a Pulse Generator generates two pulses (Sync_Short and Sync_Long) when its corresponding TCM Receiver outputs a one-tick pulse signaling the arrival of a PE synchronization message. The Sync_Short and Sync_Long pulses are delayed by a predetermined time amount after a Sync pulse to allow sufficient time for error detection and diagnosis on received messages. Let $\Pi_{SP,P3IO,RCV}$ denote the bound on the observed relative skew by the FTmr for received synchronization messages from trustworthy PEs. $\Pi_{SP,P3IO,RCV}$ is measured in units of oscillator clock ticks at the FTmr. P3IO is the process executed at the FTmr in the PE synchronization protocol as shown in Figure 4.6 in Section 4.3.1.1. The duration of the Sync_Short pulse is set to $\Pi_{SP,P3IO,RCV} + 1$. The duration of the Sync_Long pulse is $2\Pi_{SP,P3IO,RCV} + 1$. The Event Voter reads the set of trusted PEs as the eligible voters at the time it is enabled by the MC. Note that if there is an agreeing majority among the input sync events (i.e., the events are within $\Pi_{SP,P3IO,RCV}$ of one another), the middle value will be found within the time interval delimited by that majority. The outputs of the voter include the Accept signal, which is asserted to indicate that the middle event has been detected; the No_Majority signal, which is asserted when it is not true that a majority of the input sync events agree; and the Disagree_Accept(1..N), which is asserted to signal an error for each input sync event that is not within $\Pi_{SP,P3IO,RCV}$ of the middle value (and thus is not part of the agreeing majority).

Figure 11.8 shows an example of a set of three eligible voters (lanes 1, 2, and 3) and the response by the Accept() Event Voter. In this example, the maximum expected skew for pulses from trustworthy PEs is assumed to be $\Pi_{SP,P3IO,RCV} = 2$. A majority exists if 2 or more input sync events are within this skew of one another. As shown, inputs 2 and 3 are within the skew bound and form a majority. Input 1 is not part of the agreeing majority as it is three clock ticks away from its nearest sync pulse. The Event Voter detects a majority when the number of agreeing inputs is at least $\lceil (|IEV| + 1)/2 \rceil$, where $\lceil * \rceil$ is the ceiling function and $|IEV|$ is the cardinality of the set of input eligible voters. The Sync_Short pulses are used to check for agreement between the inputs, such that the overlapping pulses correspond to inputs that are within $\Pi_{SP,P3IO,RCV}$ of one another. Note the overlap of Sync_Short(2) and Sync_Short(3) in Figure 11.8. The Sync_Long pulses are used to check agreement with the middle event. In Figure 11.8, the middle event is the pulse in lane 2 as indicated by signal Internal_Accept. Agreement with this event is checked by delaying that pulse by $\Pi_{SP,P3IO,RCV}$ and determining which Sync_Long pulses overlap with the delayed Internal_Accept pulse. The Sync_Long pulses that do not overlap with the delayed Internal_Accept indicate that the corresponding input events disagree with the middle event, as illustrated in Figure 11.8 for Sync_Long(1). Based on the current SUT and FTmr designs, disagreement with the middle event is sufficient basis for an accusation against the corresponding PE.

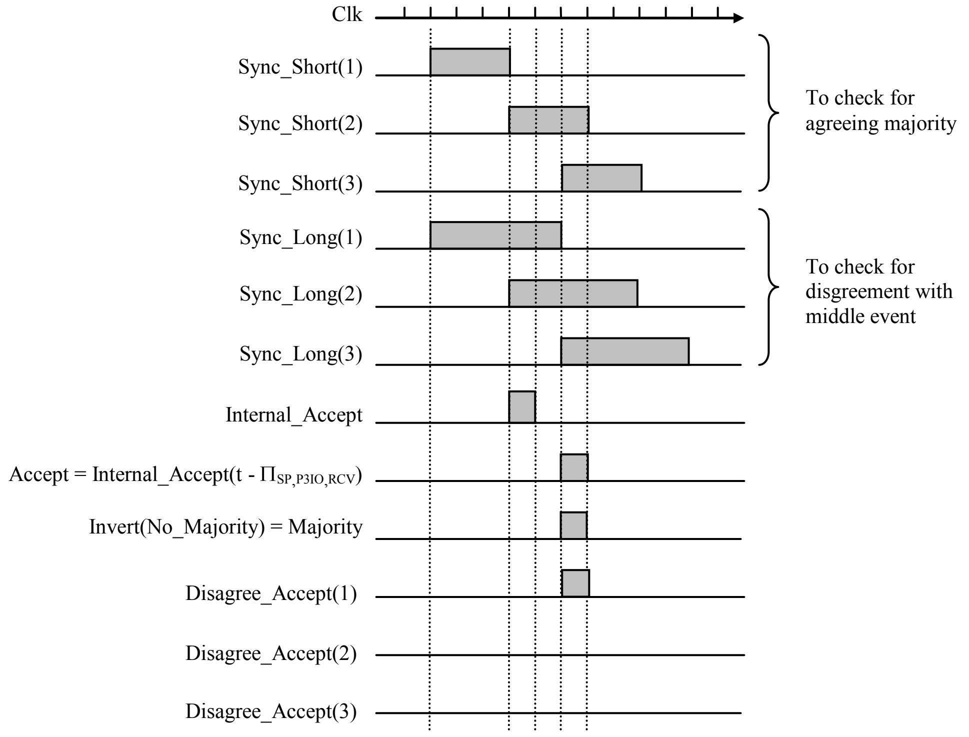


Figure 11.8: Example sync pulses and corresponding Accept() Event Voter response for $\Pi_{SP,P3IO,RCV} = 2$

11.2.2. Function Monitor

For the current SUT, in which the application consists of a simple sequence of PE message broadcasts on ROBUS, the main purpose of a Function Monitor (FMon) is to observe and report the results of receptions at a particular PE. The SFT has one FMon per PE. At the beginning of every control cycle, an FMon sends to its corresponding PE an SF_DATA TCM with the application data to be broadcast by that PE in that control cycle. After sending the data TCM, the FMon switches to a reception mode and waits for arrival of messages from the PE on what it received on ROBUS. The FMon forms and collects its observations based on what it received during the control cycle and outputs the final results at the end of the cycle. An FMon is active during a round only if its corresponding PE is active, as indicated by the Enabled_Nodes runtime configuration parameter.

Figure 11.9 shows a block diagram of an FMon connected to PTL_i to interact with PE_i. The software loads the sensors and commands data onto the buffers during the Function Setup mode. These buffers have a circular operation mode that enables the FMon to reuse their content as needed during the round. The FMon sends the SF_DATA TCM when the FTime reaches a predetermined value set at synthesis time (i.e., the transmission is a time-triggered operation). To send the TCM (see Section 7.2.2 for message format information), $FMon_i$ "prepends" a TCM tag word and a PE message header word before sending the proper application data based on the application-level role played by PE_i as indicated by the App_Assignment configuration variable. The FMon then switches to receive mode for the remainder of the control cycle. The Controller processes messages as they arrive (i.e., as an event-triggered operation) irrespective of the fact that the PE normally sends time-triggered TCMs. Thus, time of arrival is not taken into consideration in the classification of received messages. This simplifies the design and analysis of the system, but it also reduces the error detection coverage at the FMon. Time-triggered reception based on the PE communication schedule may be added as an enhancement in future version of the FMon. The processing of messages and the definition of observations is described in Section 4.3.1.2 of this report. The FMon continues operation normally as long as the FTmr remains in the Preservation mode. If the FTmr transitions to the Initialization mode, the FMon discontinues its operation as soon as processing of the current message is complete and remains idle until the FTmr recovers. The FMon does not generate an output observation report for a control cycle unless the cycle continues uninterrupted to completion. When the Round Controller issues a round-stop command, the SFT MC waits until the end of the current control cycle before disabling the FMons.

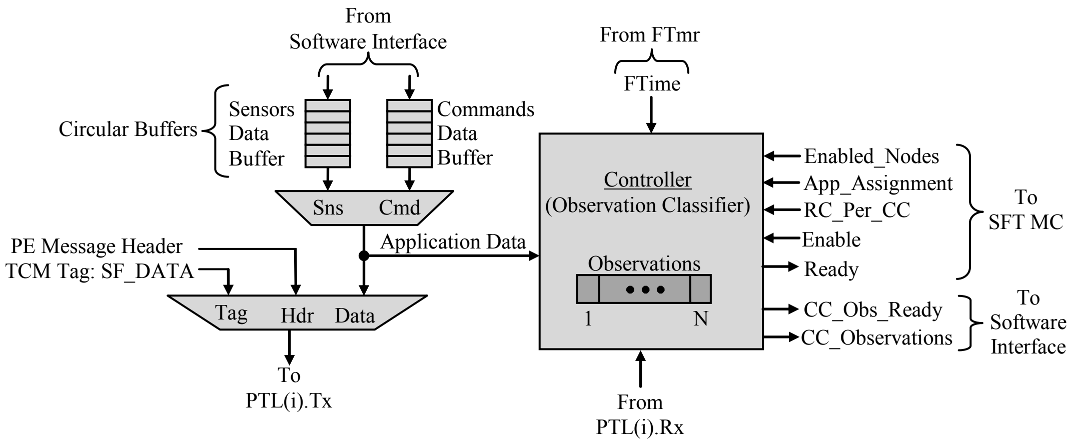


Figure 11.9: Block Diagram for the ith Function Monitor

11.3. Primary Fault-Injection Controller

The purpose of the Primary Fault Injection Controller (PFIC) is to create the fault injection messages to manage the Local Fault Injectors within the SPIDER nodes. The PFIC requires status information about the system from the Round Controller, software, and Function Timer. Each PTL for communication with the PE-BIU nodes has one sender. Another sender is allocated for the CCL to pass fault injection messages to the STC. The current implementation of the CFIMS uses the PFIC state machine in Figure 11.10 which is controlled by the signals on the interfaces shown in Figure 11.11.

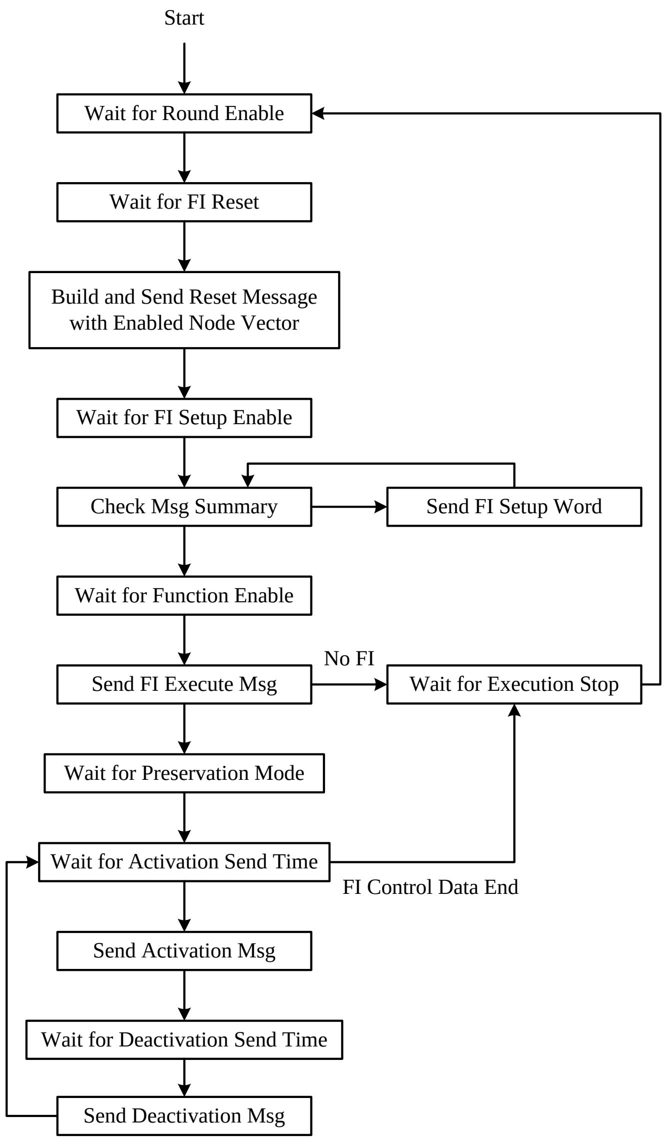


Figure 11.10: Primary Fault Injection Controller State Machine

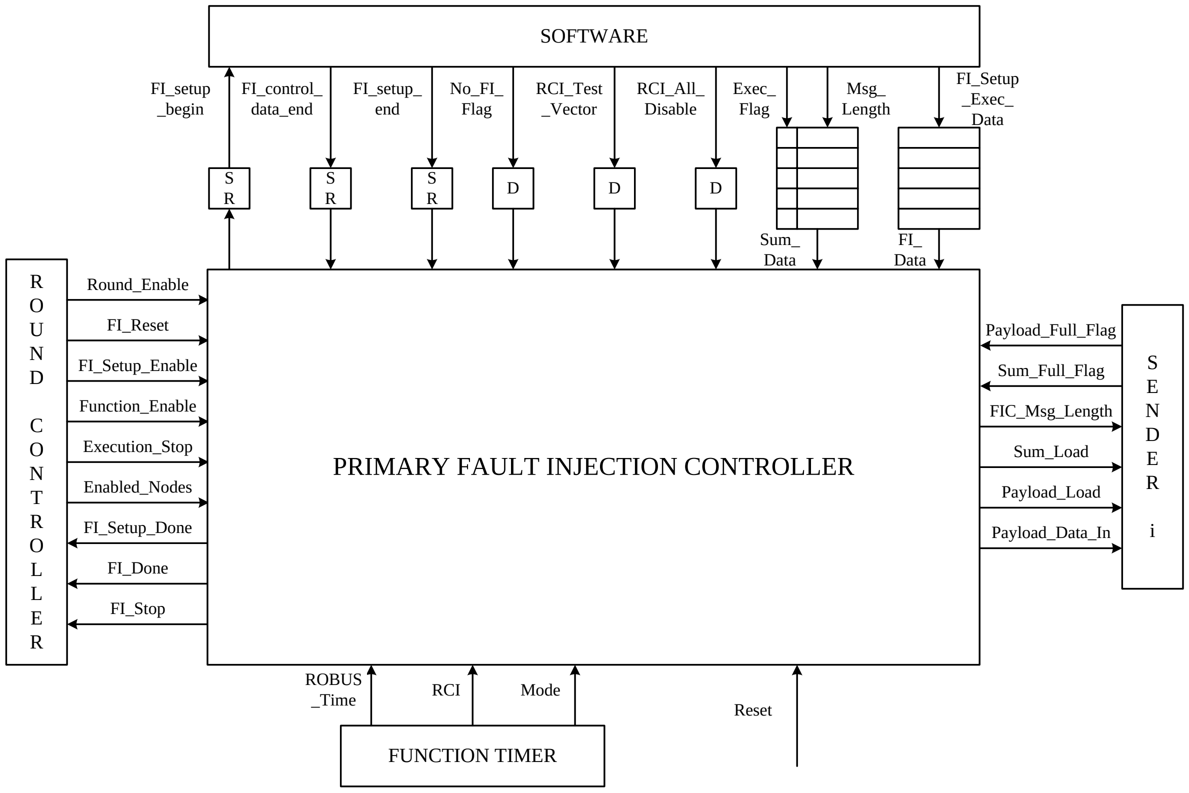


Figure 11.11: Primary Fault Injection Controller Interface Architecture

Throughout the state machine, the PFIC performs error-checking as well as polls the signal used to stop execution from the Round Controller. When an error is detected in the state machine, the PFIC stops executing and requests an execution stop to the Round Controller through the FI_Stop signal. Then the PFIC waits until the Execution_Stop has been set and cleared again before resetting the state machine. Alternatively, when another module within the CFIMS requests to stop the test, the Execution_Stop signal is set. The PFIC stops executing and resets the state machine once the signal from the Round Controller to stop execution is cleared.

At the start of a test, the PFIC waits for the round to be enabled by the Round Controller. Once received, the state machine clears the buffers and then waits for the fault injection reset signal also from the Round Controller. The PFIC then builds a TCM with a FI_Reset tag (see Section 7). This message is broadcast to the senders of all PTLs and the CCL. Within the reset message is the Node Enable Vector (NEV). The NEV has the same format as the fault vectors (see Figure A.3). Each bit signifies whether the node for that position is enabled (1) or not (0) for this particular test. The configuration section of the Test Specification file contains the NEV (see Appendix A). After the message has been sent, the PFIC waits for the start of the fault injection setup period triggered by the Round Controller. Then the software is notified the fault injection setup has begun. The PFIC obtains the length of the fault injection setup message from the Summary buffer. This length tells the state machine how many data words to pop off of the Data buffer and give to all of the senders. When the Summary buffer is empty and the software has sent notification that the fault injection setup is finished, then the PFIC waits for the Function_Enable signal from the Round Controller. When the software specified the test to have no fault injections, then the state machine waits until the end of the test for the Execution_Stop signal from the Round Controller. Otherwise, the state machine builds and sends a broadcast fault injection execute message, which allows the output of the Activation Multiplexer in the Local Fault Injectors to take function on the signal susceptible to faults (see Figure 8.2). Then the PFIC waits for the Function Timer to be in preservation mode (see Section 11.2.1) before the state machine moves on to send FIRE messages. Once the Function Timer mode is set, the PFIC waits for the RCI specified by the Test Specification file and the time within the ROBUS cycle to pop the FIRE message containing the fault vector off the buffers from the software and give the message to the senders. When the software does not signal that fault injection is finished and the FIRE message is sent, the PFIC waits for the RCI and the time within the ROBUS cycle to build and send the FIRE message to disable the faults that were injected. This disable message is identical to the FIRE message to activate the faults except that there is a data word of all zeros to deactivate the faults in place of the fault vector. Once the disable message is sent, the state machine returns to waiting for the time to activate the fault within the next control cycle. The process of waiting for the activation time of the fault through sending the FIRE disable message is repeated for each control cycle until the software signals that there is no more fault injection control data. If at this point the software buffers are empty, then the PFIC signals the Round Controller that the fault injection is done and waits for the Execution_Stop signal from the Round Controller. Then the PFIC waits for the Execution_Stop signal to be cleared to return to the beginning of the state machine and wait for the test to be enabled.

11.4. PE-BIU State Monitor

The PE-BIU State Monitor (SMon) serves two main functions: to receive the state data messages sent by the ENMs, and to determine the health status of individual PE-BIU nodes. The TCMs with SM_PE_BIU tag contain snapshots of PE-BIU state variables sent by the ENMs when triggered by node-local events. Once enabled, the SMon must monitor the arriving PTL traffic at the PTC and be ready to accept these state TCMs whenever they arrive. The state data must then be processed and forwarded to the Software Interface module for collection by the software. In addition, the SMon is required to assess the health status of individual PE-BIU nodes to contribute to the overall SPIDER-level health assessment and to provide the software (and the system user) with a simple health-based classification criterion for sifting through the state records to find evidence of fault-injection effects.

Figure 11.12 shows a block diagram of the PE-BIU State Monitor. It consists of a set of Lane State Monitors (LSM), with one dedicated monitor for every PE-BIU node, and a top-level Master Controller (MC) responsible for overall coordination and interaction with the Round Controller (RCtlr). The MC enables the LSMs when the RCtlr issues the round enable command. Only the LSMs assigned to active PE-BIU nodes, as indicated by the Enabled_Nodes configuration parameter, are actually enabled to run during the round. In addition, the SMon has the added feature of allowing the software to control which LSMs can transfer their outputs to the Software Interface module. With this feature, the software has the option of servicing the record buffers of a subset of LSMs and can prevent the other buffers from overflowing. This capability may be useful in situations where there is limited available bandwidth between the hardware processes and the software, thus requiring a selective approach for transferring data.

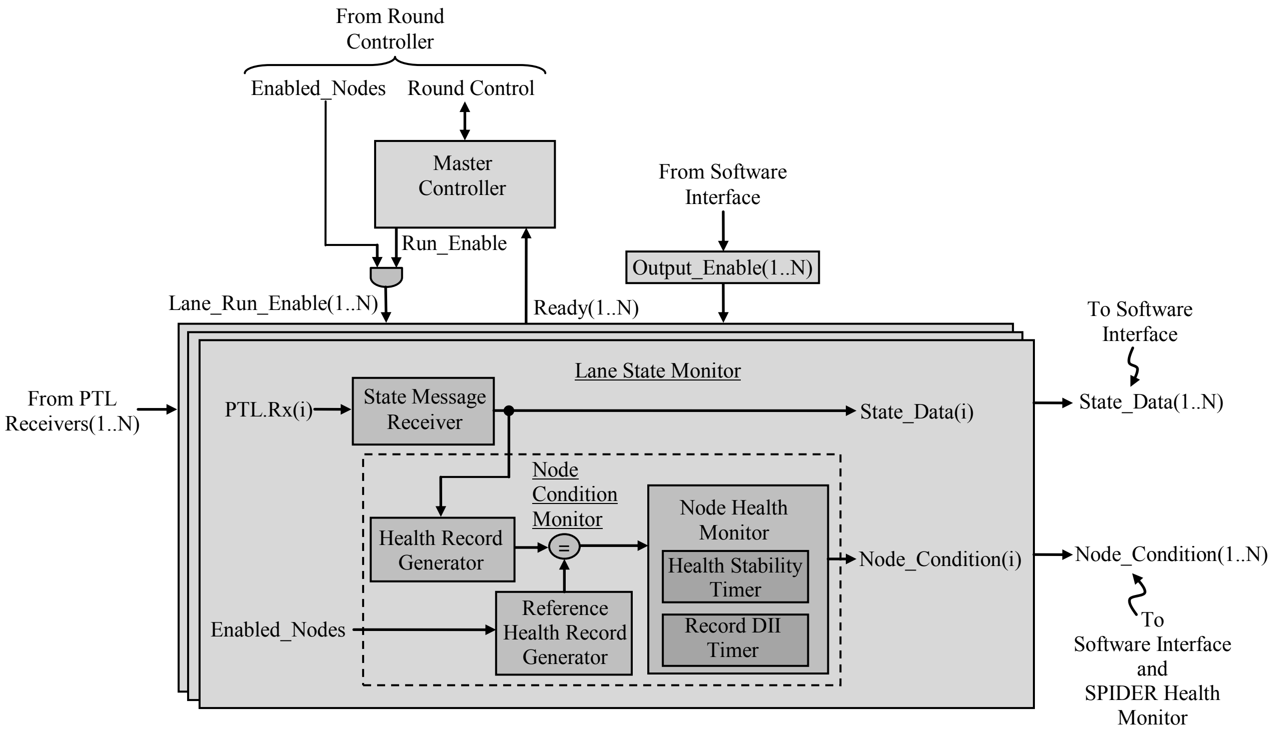


Figure 11.12: Block Diagram for the PE-BIU State Monitor

Internally, an LSM consists of a State Message Receiver (SMR) that monitors the assigned PTL for the arrival of SM_PE_BIU TCMs and a Node Condition Monitor (NCM) for assessing the health status of the corresponding PE-BIU node. The SMR's output state data is the payload section of the TCM as described in Section 7.4. The SMR can process back-to-back TCMs without an inter-message gap, which is necessary given the event-triggered operation of the ENM at the PE-BIU node. The main component of the NCM is the Node Health Monitor, which uses the state-based algorithm illustrated in Figure 11.13 to assess the status of the PE-BIU node. A Health Record contains a select subset of the variables in a state-data record referred to as the health indicators. For the current version of the system, the health indicators include the following.

- Current RPP Major Mode: Normally, the RPP operates in the Clique Preservation Mode (CPM). Execution in any other mode is due to a power-on reset or a reset triggered by the detection of a local fault or a bus failure condition (see [Torres05A]).

- RPP accusations against nodes of the opposite kind: Accusations against opposite kind nodes (i.e., RMUs in the case of a monitored BIU RPP) are due to detected errors in the direct communication links with those nodes, or due to errors in the relaying of messages as detected by the RPP suspicion mechanism (see [Torres05A]). In general, these accusations are an indication that the accused node is faulty, the accusing node is faulty, or both nodes are faulty. This can also happen if the fault assumptions of the ROBUS fault-tolerance mechanisms have been violated.

- RPP failure signal: This is a direct indication that the RPP has detected a local node failure or a ROBUS failure.

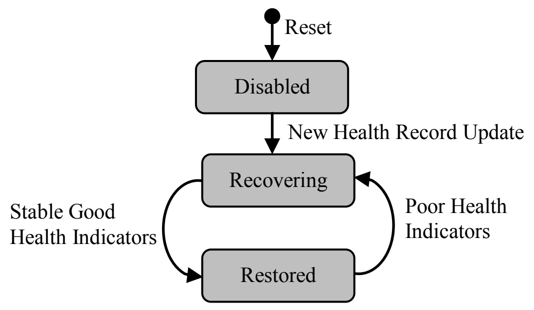


Figure 11.13: Transition diagram for state-based node health assessment

In a Reference Health Record for good health, the RPP's major mode is CPM, none of the enabled RMUs (as per the Enabled_Nodes parameter) are accused, and there is no failure reported by the RPP. As shown in Figure 11.13, after a reset, the initial assessed health state is Disabled. When a new health record arrives, irrespective of its content, the health state transitions from Disabled to Recovering. The node condition remains Recovering until consecutive good health records are received for a sufficiently long time. This is consistent with the assumption that the injected faults are always transient. The definition of the time duration for a stable good-health assessment to transition from Recovering to Restored state is a synthesis configuration parameter set on the basis on some diagnostic policy or as a result of fault effects analyses taking into consideration the assumed duration of fault-causing disturbances. In the current implementation, this parameter is arbitrarily set to two consecutive good-health records. The assessed health state transitions immediately from Restored back to Recovering whenever anything other than a good-health record is received or when the time between new health records (i.e., the data introduction interval, DII) exceeds an expected maximum value determined based on the duration of a ROBUS cycle.

When the Round Controller issues a command to stop execution of the round, the SMon MC clears the Run_Enable control signals and waits for all the LSMs to report that they are Ready to stop. The LSMs assert this signal only when they are idling and waiting for the arrival of a state TCM (i.e., the LSMs do not halt their operation in the middle of processing a TCM).

11.5. SPIDER Health Monitor

The purpose of this module is to support the Round Controller by constantly monitoring the SUT and reporting when there are relevant changes in its status. The SPIDER Health Monitor (HMon) has three phases of operation during a round. The first phase happens during the SPIDER Initialization mode, where the HMon monitors the health of the PE-BIU nodes as reported by the SMon Node Condition Monitors to detect when all the enabled nodes have reached the Restored state. This is indicative that the SUT initialization process is complete and the Round Controller can continue on to the next setup mode. The Round Controller does a timeout check on the SUT completing the initialization process and triggers a round stop if the maximum expected delay is exceeded.

The second phase of HMon operation covers the FI Setup and Function Setup major modes. As no faults are injected during these modes, it is expected that the Node Condition Monitors report that all the enabled nodes remain in the Restored state. If the condition of at least one node changes from Restored state, the HMon reports a failure to the Round Controller, which then initiates a round stop.

The third HMon phase covers the Function Execution mode. Here, the HMon monitors the operation of the FTmr as an indirect means to assess the status of the SUT. After the Round Controller issues the Function_Enable command to begin the execution, the HMon enters a wait state to allow time for the FTmr to transition from Initialization to Preservation mode. The HMon declares a failure if the FTmr has not completed the transition within the allowed time. Otherwise, the HMon monitors the FTmr mode to detect transitions back to the Initialization mode, which signal the occurrence of an SUT failure. If the FTmr mode does not return to Preservation within a predetermined time, the HMon reacts by declaring a failure because the assumptions about the characteristics of faults experienced by the SUT have been violated.

The timeout delays used in the HMon are derived from assumptions about the faults and the dynamics of system recovery. All of these delays are synthesis parameters.

The implementation of the HMon consists of a state machine with the functionality described above. The HMon has no other significant components.

11.6. Test Control Links

The Test Control Links of the PTC include the CCL for direct communication with the STC and the Primary Test Links for direct interaction with the PE-BIU nodes. The Packet Mode Communication Units (PMCUs) described in Section 6.2 are used to implement these links. In addition, the transmitters for the CCL and PTLs are coupled to multiple-access controllers (see Section 6.3) to arbitrate access to the links by multiple PTC hardware processes. The sequential design of the CCP (see Section 5) ensures that during the system configuration phase of a round there are never access collisions (i.e., instances of multiple senders simultaneously attempting to access a transmitter). For the execution phase of a round, the FTmr, FMon, and Fault Injection Controller (FIC) processes access the CCL and PTLs only on FTime-referenced triggers selected to ensure that the transmission attempts of these processes never collide. A time-triggered access pattern eliminates the possibility of TCL access interference between processes during normal operation.

The use of the priority-based access controllers at the TCL transmitters ensures that there is a predictable access control mechanism to handle event-triggered accesses and ensure that these transactions are completed without messages being dropped under most multiple-access scenarios. This simple, fixed-priority-based arbitration has the vulnerability of allowing possible scenarios of high priority sending processes indefinitely blocking lower priority processes. The PTC processes should be designed to prevent such scenarios, or to detect them and take appropriate action in case of message loss, potentially including stopping the round and reporting the incident.

At the CCL, the transmitter access ports are assigned in the following decreasing priority sequence: Round Controller, FTmr, and FIC. During normal operation in the setup and execution phases of a round, none of these process attempt to access the CCL transmitter simultaneously. The Round Controller is given highest priority because during the Function Execution phase it accesses the CCL only to transmit an RC_STOP message to end a round. The transmission of a FTmr or FIC message after an RC_STOP is inconsequential. In the execution phase, the FTmr's SYNC_ROUND_TIME messages carry time-critical information that must have a known bounded communication delay with the least possible delay uncertainty. The FIC is allowed to access the CCL whenever the Round Controller and FTmr are not accessing the link. The FIC should be designed to ensure that there is never a collision between FTmr and FIC messages.

At the PTLs, the assigned access priority, in decreasing order, is: FTmr, FIC, and FMon. The FTmr is given highest priority because of the timing-critical nature of its messages. The FIC messages are normally very short and generally require higher propagation precision than FMon messages. The FMons operate on a control cycle time scale and have longer and less time-sensitive messages.

11.7. Software Interface

The purpose of the Software Interface module is to serve as a communication bridge translating the format and timing of control, status, and data signals between the PTC functional processes implemented as custom hardware circuits running on the FPGA and the functional processes implemented as code segments on a software program running on the CPU. On the RSPP1 nodes, the communication link between the FPGA and the CPU is an ISA bus (see Section 4.2.1). Data transfers on the bus are initiated by the CPU, with the timing of the transactions being influenced by the application program, the operating system, and the hardware layer of the CPU board. The transactions on the bus consist of 16-bit read or write accesses. The FPGA board is configured with an ISA interface of sixteen 16-bit read and write ports. All communication between the PTC hardware and software processes is constrained to happen through this interface. The Software Interface module handles the conversion between the custom data types in the hardware and the fixed-width read and write ports of the ISA interface.

Figure 11.14 shows a block diagram for the Software Interface module. This module consists of two main sections: the multiplexing-demultiplexing (mux-demux) interface (to the right) between the ISA and the non-multiplexed read and write ports, and the functional interfaces (to the left) that interact directly with other PTC hardware processes. The mux-demux ISA Interface is a generic module developed in a previous project. The functional interfaces are custom modules that realize an agreed-upon detailed interface between the hardware and software processes of the PTC.

The Round Controller Functional Interface handles the direct interaction between the Round Controller and the software processes for the implementation of the CCP, as described in Section 5.

The Round Timer Functional Interface is a simple buffer for records of significant events in the evolution of the RTmr value during the round. The goal is to faithfully regenerate the RTmr timeline during post-test processing. This is accomplished by recording the value of the IT and IC elements of the RTmr at the end of every time interval when the IT holds the duration of the interval and just before the IC is incremented. The purpose of the RTmr Record Writer is to package the data into records with a particular format for use by the software. The configured depth of the buffer is dependent on the maximum record input rate (which is determined by the ROBUS cycle duration) and the bound on the longest time interval that records will accumulate in the buffer while the software is busy tending to other tasks. It is assumed that during a read burst the software is capable of reading the record buffer much faster than the input rate. The Round Time is shared with other functional interface modules for use as the record time tag.

The Fault Injection Controller Functional Interface consists of Control and Status Interface logic for transferring data and signals. All the buffering required for setup and injection of faults is done locally at the FIC process (see Section 11.3).

As in the case for RTmr, the Functional Timer (FTmr) Functional Interface consists of a simple buffer to record significant events in the FTime. In addition to allowing, a recreation of the FTime timeline during the round, the FTmr state information included in the records provides insight into the status of the SUT and the effects of injected faults. An FTmr record is created when the FTime reaches the end of a ROBUS cycle or when the operational mode of the FTmr transitions from Preservation to Initialization.

Both of the FMon and SMon functional interface modules contain one record buffer per PE-BIU node. The interface modules also contain additional control and status logic to setup and manage the operation of these hardware processes.

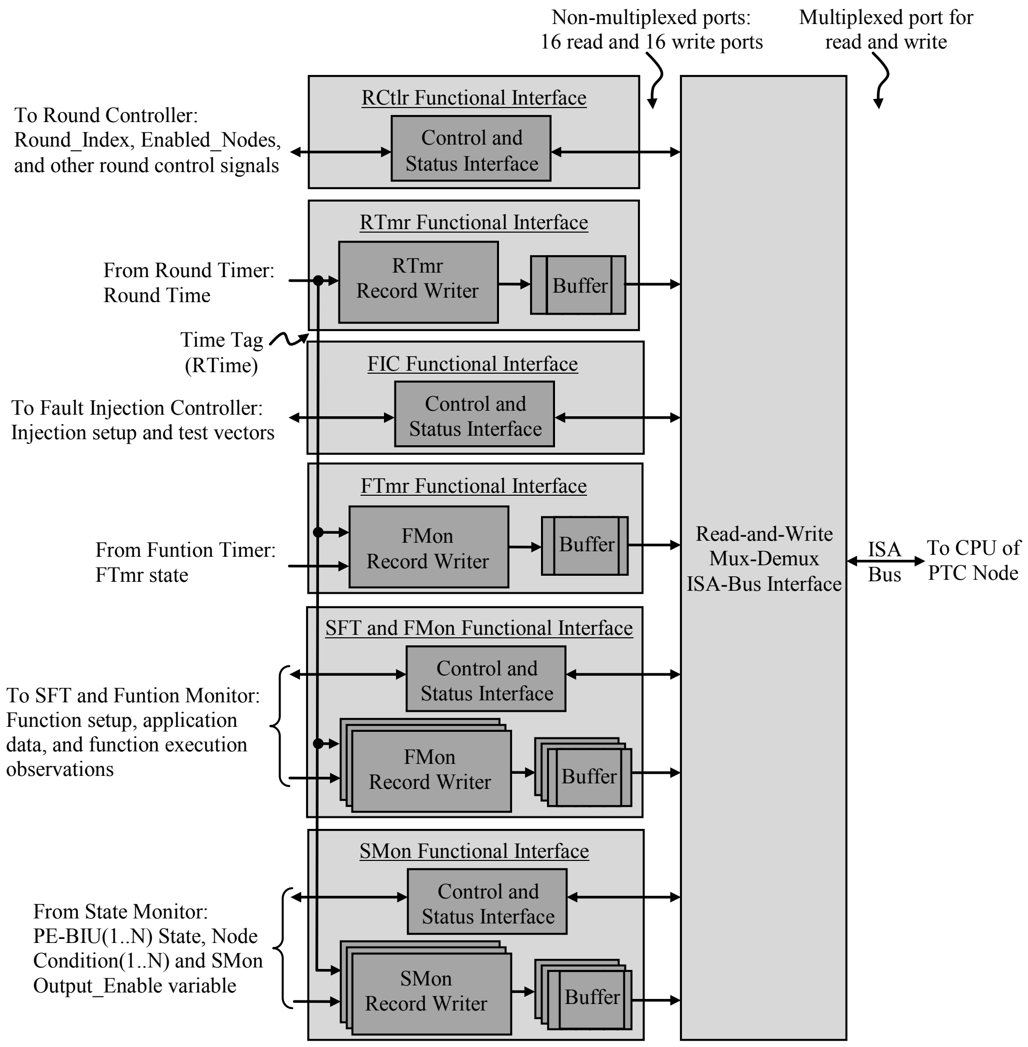


Figure 11.14: Block diagram for the PTC Software Interface module

12. Hardware Processes of the Secondary Test Controller

The main reason for having multiple test controllers is to support SUT configurations with a total number of nodes larger than the number of custom communication ports available in one RSPP node (currently, only eight ports). As currently configured, the PTC handles all direct communication with the PE-BIU nodes, and the STC interacts directly with the RMUs. At a higher level, the PTC and STC coordinate their actions with the goal of behaving as a single large controller from the viewpoint of the SUT.

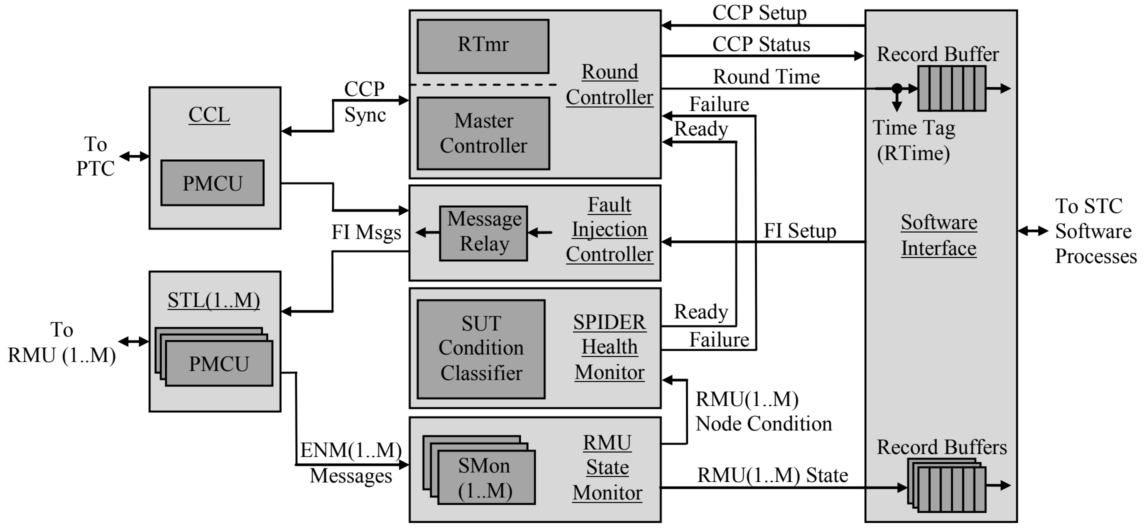


Figure 12.1: Block diagram for the Hardware Processes of the Secondary Test Controller

Figure 12.1 shows a block diagram for the STC hardware processes. Compared to the PTC as described in Section 11, the STC has a much simpler design as there are no processes to interact with PE-level service protocols and the SUT application. The STC Round Controller (RCtlr) has essentially the same design as the PTC RCtlr, but it is slightly simpler as it interacts with fewer processes. The STC RTmr also behaves similarly to the one at the PTC, with the exception that during the Function Execution mode of the CCP (see Section 5), it synchronizes to the FTmr by means of the SYNC_ROUND_TIME TCMs (i.e., the "Sync" messages) received from the FTmr over the CCL. The STC Fault Injection Controller (FIC) is configured as a simple function for relaying Primary Fault Injection Controller (PFIC) messages received over the CCL and sent out to the RMUs on the Secondary Test Links (STLs). The RMU State Monitor (SMon) receives state messages sent by the ENMs at the RMUs and forwards their content to the Software Interface, where they are buffered until the software can read them. The SMon also contains Node Condition Monitors (NCMs) to assess the health status of individual RMU nodes. These node health assessments are sent to the SPIDER Health Monitor process, whose function is to support the RCtlr by monitoring the health of the SUT.

The following subsections provide additional information about the STC hardware processes. The descriptions are kept brief as most of the necessary insight has already been given with the presentation of the PTC hardware processes in Section 11.

12.1. STC Round Controller

The operation of the STC RCtlr is similar to the operation of the PTC RCTlr. The differences are reflected in the design of the CCP as described in Section 5. A block diagram of the STC RCtlr is shown in Figure 12.2. Structurally, the STC RCtlr and PTC RCtlr (see Figure 11.2) are identical. The only significant difference is that the RTmr accepts Sync messages received over the CCL from the FTmr. These messages are used to synchronize the RTime during the Function Execution mode of the CCP. Additional information about the RCtlr can be found in Section 11.1.

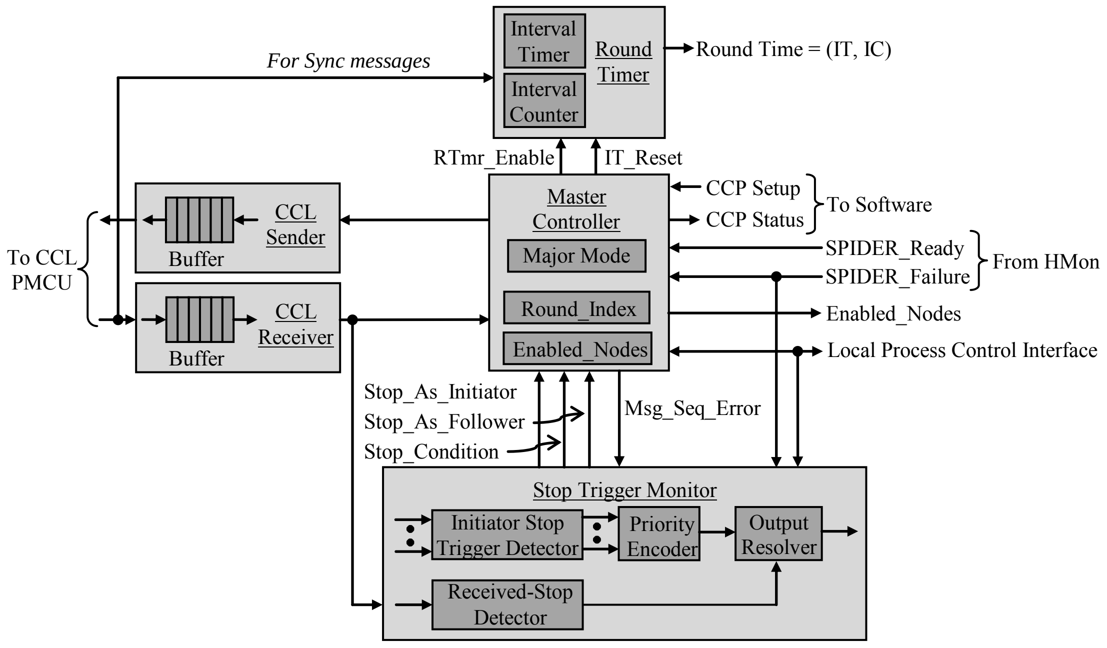


Figure 12.2: Block diagram for the STC Round Controller

12.2. Secondary Fault-Injection Controller

The purpose of the Secondary Fault Injection Controller (SFIC) is to pass messages from the PFIC to the RMUs. A PFIC sender connected to the CCL sends messages to the STC. The SFIC monitors the incoming messages on the CCL and forwards the fault injection messages on to the STLs for the RMUs. The senders in the SFIC are only connected to the STLs because the PFIC does not need any information from the SFIC through the CCL. The SFIC also has the responsibility of updating the status of the fault injection to the Round Controller and software. The two independent tasks of the SFIC led to the design of two separate, concurrent state machine processes. The Relay State Machine handles the forwarding of messages to the RMUs, while the Control Signal State Machine handles the control of the fault injection by interfacing with the Round Controller and software. Similar to the PFIC, both of these state machines perform error-checking, and the Control Signal State Machine polls the signal used to stop execution from the Round Controller through the FI_Stop signal shown in Figure 12.3.

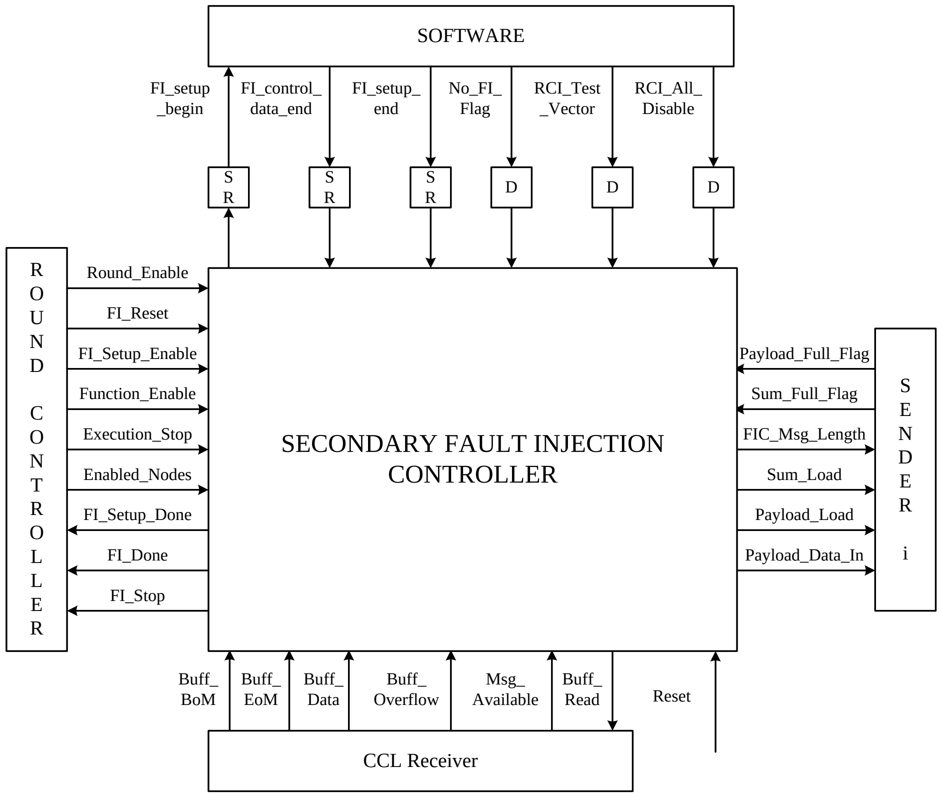


Figure 12.3: Secondary Fault Injection Controller Interface Architecture

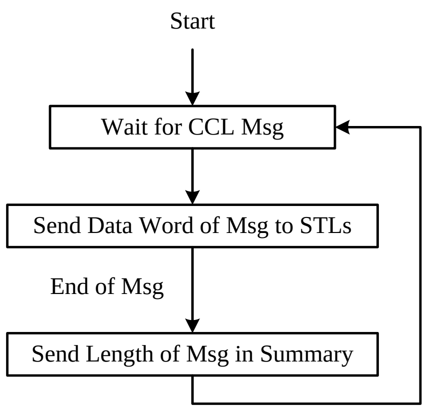


Figure 12.4: Secondary Fault Injection Controller Relay State Machine

The Relay State Machine, shown in Figure 12.4, monitors the CCL for messages from the PFIC that need to be forwarded to the RMUs. When a CCL message is available, the state machine sends each data word of the message to the STLs via the Payload_Data_In signal of each Sender i. It specifies that it is part of the message by setting the sender Payload_Load signal. Once the entire message has been sent, the state machine sends the summary data word containing the length of the message that was just sent. This time the Sum_Load signal is set instead of the Payload_Load. After the summary is sent, the Relay

State Machine returns to wait for another CCL message. This process repeats for all CCL messages unless there is an error.

Throughout the execution of a round, the Control Signal State Machine, shown in Figure 12.5, handles all of the control signals used by the PFIC in the interfaces with the software and Round Controller. At the beginning of a round, the state machine waits for the Round Enable signal to be set by the Round Controller before it resets its buffers. Then it waits for the FI_Reset and the FI_Setup_Enable signals from the Round Controller to be set before it sets the FI_setup_begin signal to the software. Once the software completes the fault injection setup phase, it sets the FI_setup_end flag. When the PFIC state machine sees FI_setup_end set, it sends the FI_Setup_Done signal to the Round Controller. After the Function_Enable is set by the Round Controller, the Control Signal State Machine waits for the stop condition from the Execution_Stop signal of the Round Controller to become true before it sets up for the next round.

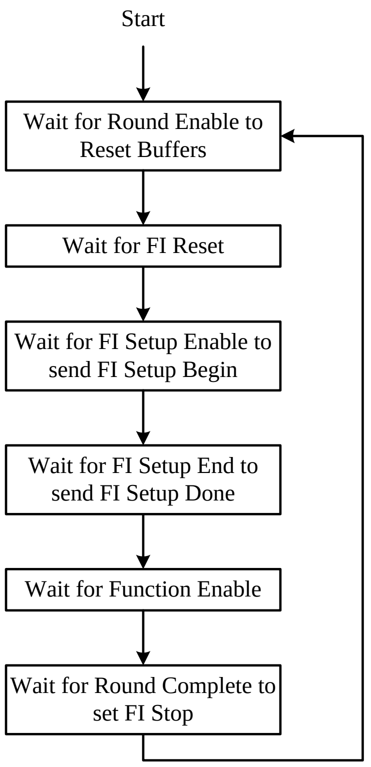

Start
Wait for Round Enable to Reset Buffers
Wait for FI Reset
Wait for FI Setup Enable to send FI Setup Begin
Wait for FI Setup End to send FI Setup Done
Wait for Function Enable
Wait for Round Complete to set FI Stop

Figure 12.5: Secondary Fault Injection Controller Control Signal State Machine

12.3. RMU State Monitor

The function and implementation of the RMU SMon are similar to the ones for the PE-BIU SMon at the PTC. The SMon is responsible for receiving the SM_RMU TCMs from the RMU ENMs, forwarding the state data to the software interface, and monitoring the health of the RMU nodes. Additional relevant information about the operation and implementation of this module can be found with the description of the PTC's PE-BIU SMon in Section 11.4.

12.4. Health Monitor

The STC Health Monitor (HMon) covers two phases of operation in a round: SPIDER initialization and system setup. In the SPIDER Initialization phase, the HMon signals to the RCtlr that SPIDER is ready when the SMon reports that all the RMU nodes are in the Restored state. In the next phase of operation, which continues until the CCP enters the Function Execution mode, the HMon reports an SUT failure if the SMon reports that the health condition of at least one of the RMUs has transitioned back to the Recovering state, even if it did so momentarily. The STC HMon is not active in the Function Execution mode as it does not have access to any additional relevant SUT information beyond what the PTC monitors.

As is the case for the PTC HMon, the implementation of the STC HMon consists of a single state machine with no other major structural elements.

12.5. Test Control Links

The STC TCLs have similar desired access attributes as the PTC TCLs. However, the simpler design of the STC simplifies the TCL access problem. Currently, the RCtlr and the FIC are connected to the CCL transmitter, but only the RCtlr is actually sending any messages on the link. The RCtlr is given higher access priority than the FIC as the only time a simultaneous access may be attempted is on the transmission of an RC_STOP message by the RCtlr, which makes additional FIC messages irrelevant as the RC_STOP message is only sent to finish a round. The connection by the FIC is a provision for future expanded fault-injection capabilities requiring message flow from the STC to the PTC. At the STL, only the FIC is connected to the transmitter.

Additional information about interfacing to the TCLs can be found in Section 11.6.

12.6. Software Interface

The function of the STC Software Interface module is a subset of the function by the PTC Software Interface. The main difference is the absence of the SFT functional interfaces for the FTmr and FMons. Figure 12.6 shows a block diagram for the STC Software Interface. Additional information about the component modules can be found in Section 11.7.

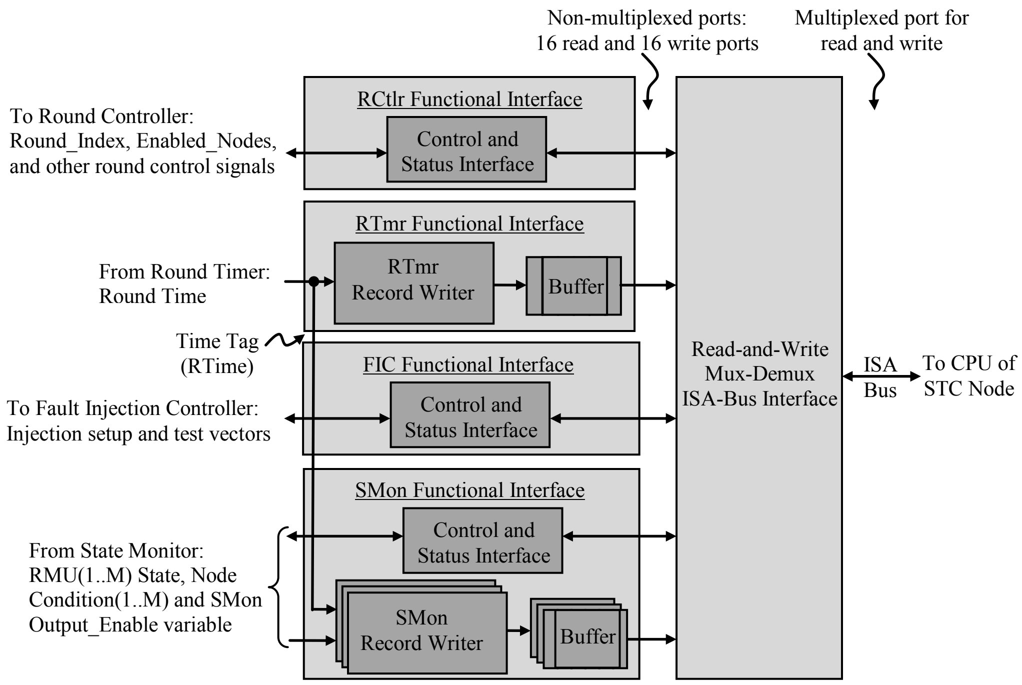

Non-multiplexed ports: 16 read and 16 write ports
Multiplexed port for read and write
RCtlr Functional Interface
Control and Status Interface
To Round Controller: Round_Index, Enabled_Nodes, and other round control signals
RTmr Functional Interface
RTmr Record Writer
Buffer
From Round Timer: Round Time
Time Tag (RTime)
FIC Functional Interface
Control and Status Interface
To Fault Injection Controller: Injection setup and test vectors
SMon Functional Interface
Control and Status Interface
From State Monitor: RMU(1..M) State, Node Condition(1..M) and SMon Output_Enable variable
SMon Record Writer
Buffer
Read-and-Write Mux-Demux ISA-Bus Interface
ISA Bus
To CPU of STC Node

Figure 12.6: Block diagram for the STC Software Interface

13. Test Control Software

The Test Control Software consists of several programs implemented in C to provide an interface between the test operator and the hardware of the SUT. The software communicates with the hardware via the Software Interface module of each Test Controller described in sections 11.7 and 12.6. Compared to hardware processes, the software processes have a much more coarse timing. The Software Interface translates the format and timing of control, status, and data signals between the software and the RCtlr, RTmr, FIC, FTmr, SFT, FMon, and SMon of the PTC and STC hardware. Each Test Controller (PTC and STC) contains its own software program running on separate CPUs as shown in Figure 13.1. These two Test Execution Software programs have separate but closely related functional processes because of the different roles the PTC and STC have in the system.

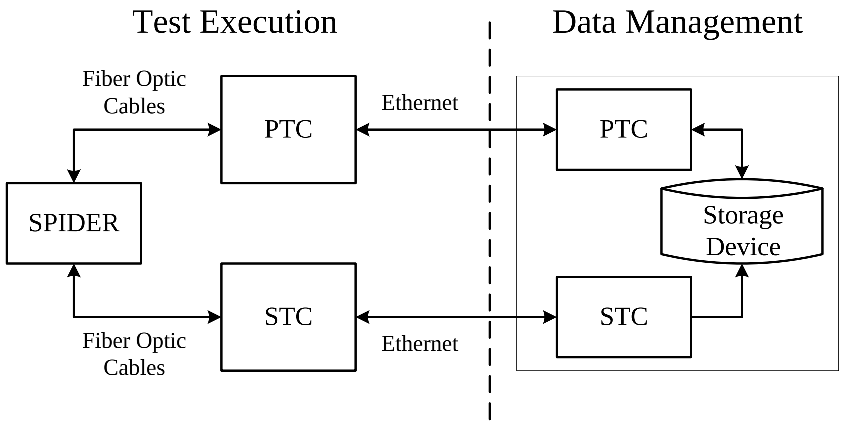


Figure 13.1: Separation of the Test Execution and Data Management Software

The Test Specification file (see Appendix A) contains a setup and an execution section. The setup section consists of system configuration parameters for the SUT and the faults to be injected, whereas the execution section consists of the list of fault vectors to be injected on the control cycles. Since the PTC hardware forwards setup messages to the STC via the CCL, only the PTC Test Execution Software has the Configuration Management process to decipher the Test Specification file setup section and pass the configuration setup to the hardware. During the test, the fault vectors in the execution section of the Test Specification file are supplied by the software to the hardware. Only the PTC Test Execution Software needs to read and send the fault vectors because of the automatic forwarding built into the PTC hardware. Therefore, only the PTC is required to have any interaction with the Test Specification file. However, both the PTC and STC Test Execution Software programs have the responsibility of collecting execution data for post-test analysis, providing real-time monitoring for the test operators, and starting and stopping a round as determined by the test operator.

13.1. Test Execution and Data Management

The amount of I/O data is directly proportional to the number of control cycles, i.e., duration of a round. Data management is needed to handle the considerable amount of I/O data for typical test durations. Pre-test data is generated before the round execution, while observation data is generated during the round execution. The Test Specification file contains all of the pre-test data including data to establish the setup configuration of the system and one 16-bit fault vector for each control cycle of the round. The observation data includes all of the data described in Appendix B. The condition of the system that caused the round to stop and error reports for the round are recorded in the test log. The RTmrs, FTmr, FMons and SMons report their respective observations to the software. In order for the observation data to be available post-test, it is written to files for permanent storage and future analysis.

Since the PTC and STC Test Execution Software programs are located on nodes with a PC/104 stack described in Section 4.2, the RAM and flash memory of the node are not adequate for storage of the data. The nodes do not have enough room to store multiple test rounds of output files in their flash memory. Periodically, the output files would have to be transferred to a permanent storage device to allow space in the flash for new rounds. The constant rewriting to the same location in flash memory on the nodes could potentially damage the memory device. To prevent rewriting to a file on the nodes, the data is stored in RAM on the nodes during the test and then transferred to the permanent storage device after the execution phase of the round. The storage device requires a Data Management Software to read and write the Test Specification file and observation files, respectively. Data is passed between the Data Management Software and the Test Execution Software, as illustrated in Figure 13.1, using Ethernet sockets for the communication channels. Sections 13.2 and 13.3 describe the high level Data Management and Test Execution Software, respectively, while the detailed pseudo-code of all of the software is found in Appendix C.

During the test, pre-test data is read from the input Test Specification file and observation data is written to an output file for post-test analysis. File I/O is known to take a significant amount of time. To avoid time-critical operations being delayed because of file I/O, the Test Control Software does not perform any file interaction during the execution of a round. The software reads the entire Test Specification file prior to starting a round and stores the contents in volatile memory (RAM). During the round, the software stores the data it gathers from the hardware in RAM. A key benefit of using the software for the data management is that it typically has access to a larger amount of RAM. At the end of the test execution, the software transfers the observation data from RAM to files on storage devices.

The modules in Figure 13.1 are built on a variety of devices. As described in Section 4.3, each functional PE-BIU and RMU node in the SPIDER module of the diagram run on separate RSPP physical nodes. The functionality of the SPIDER nodes is located on the FPGA board of the PC/104 stack (see Section 4.2.2). The PTC and STC are located on separate RSPP physical nodes with both hardware-implemented functions on the FPGA and the Test Execution Software on the processor. Since the RSPP nodes have a White Dwarf Linux operating system, the Test Execution Software is compiled with a Linux gcc compiler. The PTC and STC nodes are connected to a separate desktop PC (Personal Computer) used for data management. The Data Management Software is compiled on the PC with a Windows operating system using a gcc compiler on the Cygwin Linux-like environment. The shared PC runs the PTC and STC Data Management Software processes during a test. The hard drive of the data management PC acts as the storage device during a test. The operator copies the files periodically to an external hard drive connected to the PC that is used as a backup storage device.

The data format for socket communication and permanent storage influences the speed of transmission and size of the data on the storage device, respectively. All observation data, with the exception of the test log, is stored in RAM as 16-bit unsigned short integers. If the observation data was converted to text format and then sent to the Data Management software, each character would take 8 bits, which would be a total of 136 bits for the 16-bit word and a newline character. Therefore, the bulk of the observation data is sent using the unsigned short integer format, which only requires 16 bits for each data entry. This is about 8.5 times faster than transferring in text format. Initially, the entries of the observation data were stored to text files in 16-bit binary format. However, the amount of space required on the hard drive for a binary file format using unsigned short integers is much less than using the text file version. Similar to the transmission, the observation data storage of the Data Management software uses binary file types rather than text for the hardware observation files. Since less space is required for the binary format, the files are more compact. A complication with the binary format for the files is that it is not readable; however, the binary files containing the observation data can be post-processed in various formats to make the analysis simple.

The PTC and STC software processes provide the high level test execution and data management of the CFIMS. Figure 13.2 depicts the sequence of data transferred between the software processes in relation to the execution of a round. The repository containing the storage device with all the I/O data uses a Windows XP operating system. The PTC and STC Data Management Software programs are a part of the data repository and are located on the same machine as the storage device. The PTC program deciphers the Test Specification file and sends the runtime parameters and fault vectors via Ethernet to the Test Execution Software located in the PTC node. The PTC Test Execution Software configures the hardware. The Test Execution Software associated with both test controllers receives the observation data from the hardware and stores it in RAM until the end of the round. Then the corresponding Data Management Software writes the received observation to the storage device.

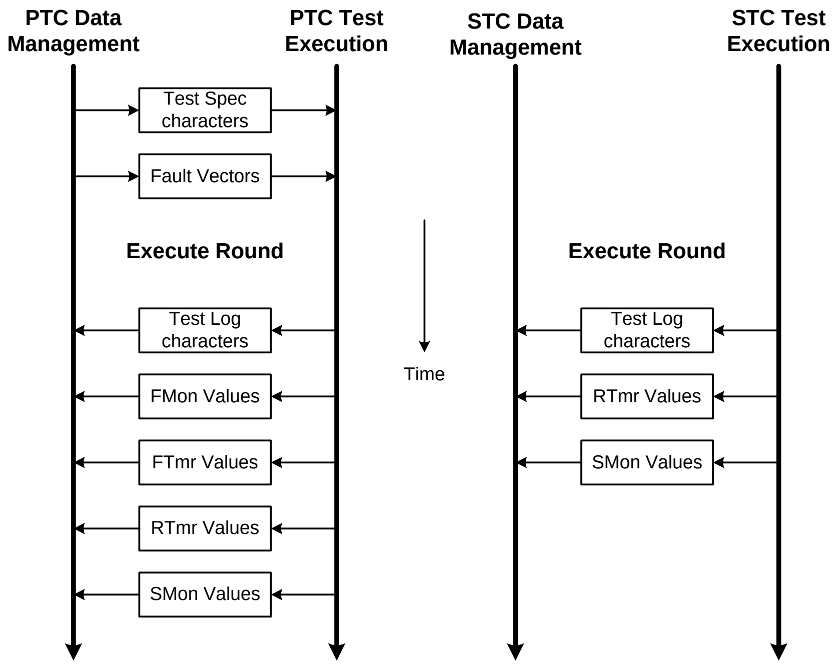


Figure 13.2: Test Execution and Data Management Software Data Transfer

Each block of Figure 13.2 represents the data transmission protocol described in Figure 13.3. Each horizontal arrow in Figure 13.2 represents the transmission of data from the transmit side to the receive side of Figure 13.3. The data transmission protocol was developed to ensure no data is lost during the socket transfer. Socket transfer has built-in CRCs for error detection in transmission. Therefore, the protocol only needs to check for the omission of data being transferred, or when data is dropped during the transmission. During the exchange of setup information at the beginning of a round, the roles of the Transmitter and Receiver are associated with the Data Management and Test Execution, respectively.

Once the round of execution is over, the roles are reversed to transfer the results. Before sending the data, the Transmitter counts how many data words will be sent and transmits this count, called the Tx Count, to the Receiver. Then the Transmitter sends each data entry and waits to receive a count response from the Receiver. The Receiver counts the data entries as they arrive in a Rx Count variable. Once the Rx Count at the Receiver matches the Tx Count, the Receiver sends the Rx Count to the Transmitter. The transmitter then compares the Tx Count and the Rx Count. If they are equal, the data transfer is considered a success. Otherwise, the Transmitter will consider the transfer a failure. If a failure occurs in the transfer of Test Specification file data, the round will not execute and an error is reported. If a failure occurs in the transfer of observation data, the data on the transmit side is stored in a file to be transferred post-test. Each transmission has a timeout associated with it at the receiving end of the arrow. If the data does not arrive within the allotted time-window, the transfer is considered a failure and is treated the same way as the Tx Count not equaling the Rx Count above. The duration of the timeouts of each data word are one second because the data should be transmitted back-to-back. The timeout duration for the Rx Count is also relatively short because the Receiver will send it immediately after receiving the last data word. However, the duration of the Tx Count timeout will vary depending on what data is to be transferred. For example, the PTC Data Management Software will wait indefinitely for the test log character Tx Count because the duration of the round may vary.

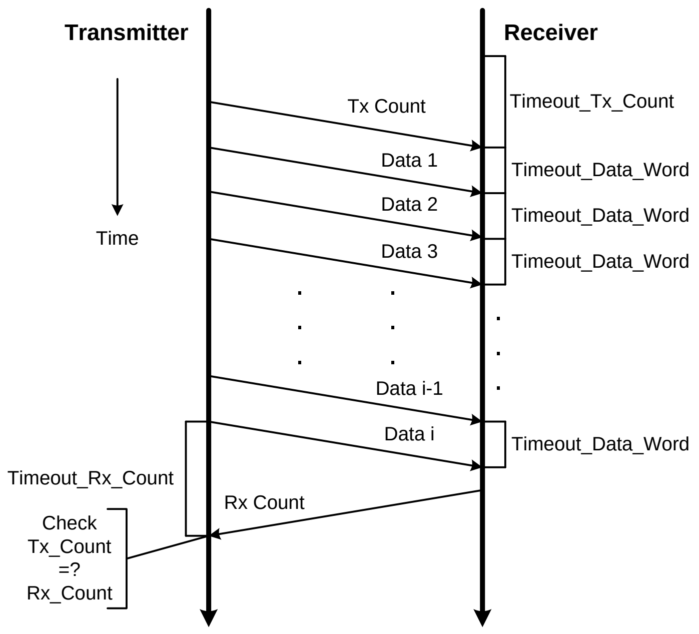


Figure 13.3: Test Execution and Data Management Software Data Transmission Protocol

13.2. Data Management Software

The Data Management Software transmits the configuration and test flow control input data and stores the observation data results for analysis. The PTC and STC follow the same high level program flow of the Data Management Software as shown in Figure 13.4. At the start of the program, the system is initialized by setting up the communication socket with the Test Execution Software located on the physical PTC and STC nodes. The initialization phase on the PTC Data Management Software also includes reading the Test Specification file and sending the contents to the PTC Test Execution Software. Then the PTC and STC Data Management Software monitor the socket waiting for data from the Test

Execution Software. Since the entire round duration must be completed before the Test Execution Software sends any observation data over the socket, the Data Management Software does not have a timeout for the first piece of observation data. When data is received at the Data Management Software, it is written to the appropriate observation file (see Appendix B for the observation data). Figure 13.2 shows that the transmission of the PTC observation data always starts with the Test Log data followed by the FMon, FTmr, RTmr, and SMon data. The STC observation data follows the same order as the PTC without the FMon and FTmr data. If there is another block of observation data to be received, the program returns to monitoring the socket for that data. Once all of the observation data has been received and written to file, the Data Management Software closes all of the files and the socket before ending the program.

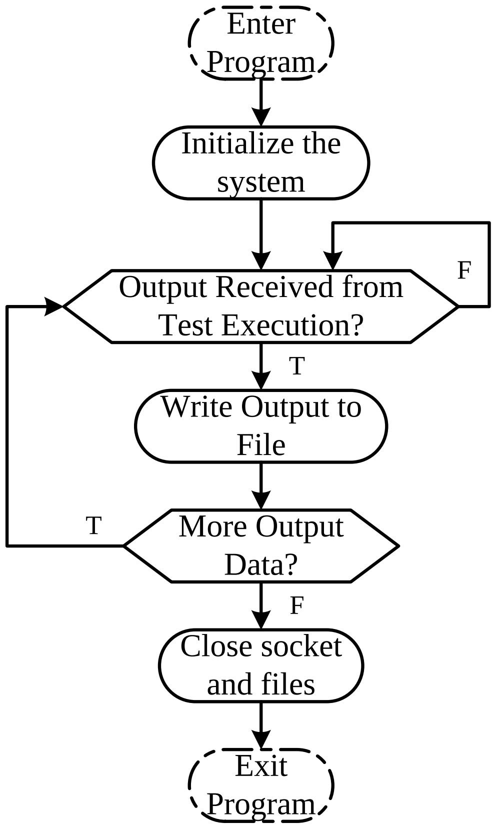


Figure 13.4: PTC and STC Data Management Software High Level Program Flow

13.3. Test Execution Software

The Test Execution Software interacts with the Data Management Software for data storage and with the hardware of the CFIMS to control the test execution. The PTC and STC follow the same high level program flow of the Test Execution Software as shown in Figure 13.5. At the beginning of the program, the system is initialized. The FPGA is configured to use the appropriate hardware file for either the PTC or the STC. The system is reset to clear out any uninitialized data before the start of the test. The socket to the Data Management Software is configured. The PTC Test Execution Software also receives the setup and execution sections of the Test Specification file to configure the system from the socket to the Data Management Software. The monitor of both the PTC and STC Test Execution Software programs is set up with the initial values to give the operator insight into the test. After the hardware system is reset, it remains idle until the Round_Begin signal is sent from the Test Execution Software. The software waits for the operator to press a key on both controllers signaling the start of a round. Once a key is pressed, the software sends the Round_Begin signal to the hardware, which gives the hardware control of the round while the software mainly tracks the progress of the hardware round. The Test Execution software enters the main loop where it follows the hardware execution waiting for the declaration that the round is finished. Throughout the round, when the Data Collection processes of the Test Execution Software see there is available observation data from the hardware, the software gathers the observations about the operation of the SUT and the CFIMS and stores the data in RAM until the end of the round. At the end of the round, all of the observation data stored in RAM is then transferred to the Data Management Software, where the data is written to file for post-test analysis.

If there is no available observation data during iterations of the Test Execution Software main loop, then it checks a half-second timer. Every half second, the software alternates between checking the keyboard for a manual stop by the operator and updating the status information on the screen of the monitor. Thus, each task is done once per second during a round. If there is no data available and it is not time for the screen or keyboard to be updated, the software will execute the CCP State Machine, which coordinates the interaction of the hardware and software during the setup and execution of a round. The CCP State Machine for the PTC is more complex than that of the STC since the STC has a much more passive role in round execution.

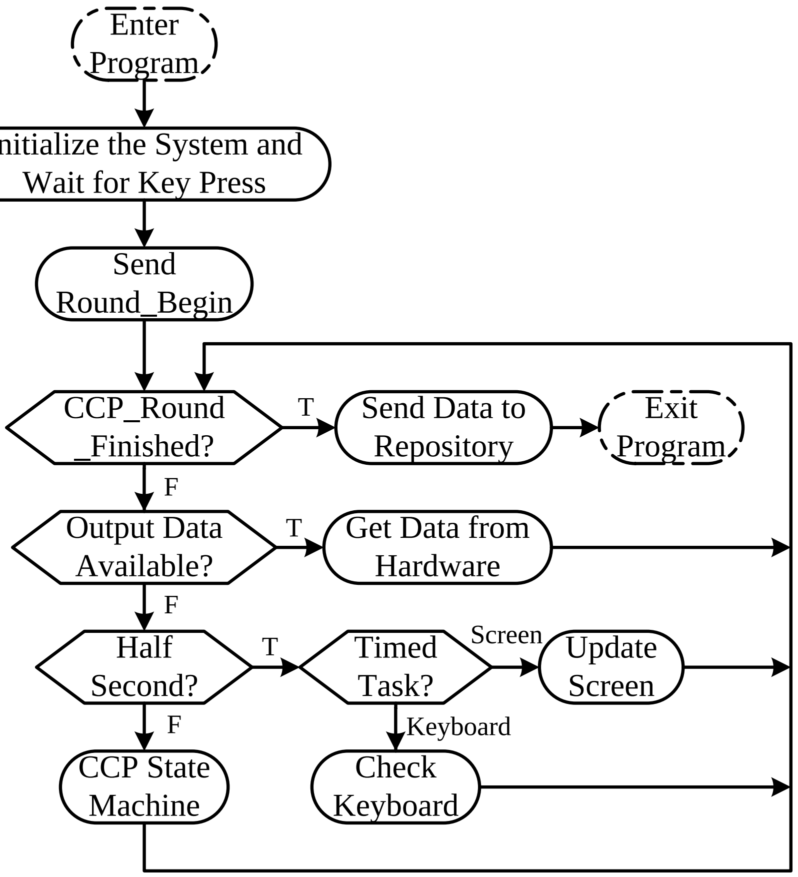


Figure 13.5: PTC and STC Test Execution Software High Level Program Flow

Figure 13.6 illustrates the CCP State Machine for the PTC. The execution of the state machine follows closely the event sequences of Section 5.1. All of the states except the final FINISH_UP_ROUND state check for a stop condition (see Figure 13.7) before they execute. If the hardware is stopped in any CCP state, the state machine transitions to the FINISH_UP_ROUND state where the final observations are recorded and the round ends. If there was an overflow in the observation data being received from the hardware or the operator presses a key, a signal is sent to the hardware signifying the end of the round and the state machine transitions to the WAIT_HW_ROUND_STOP state.

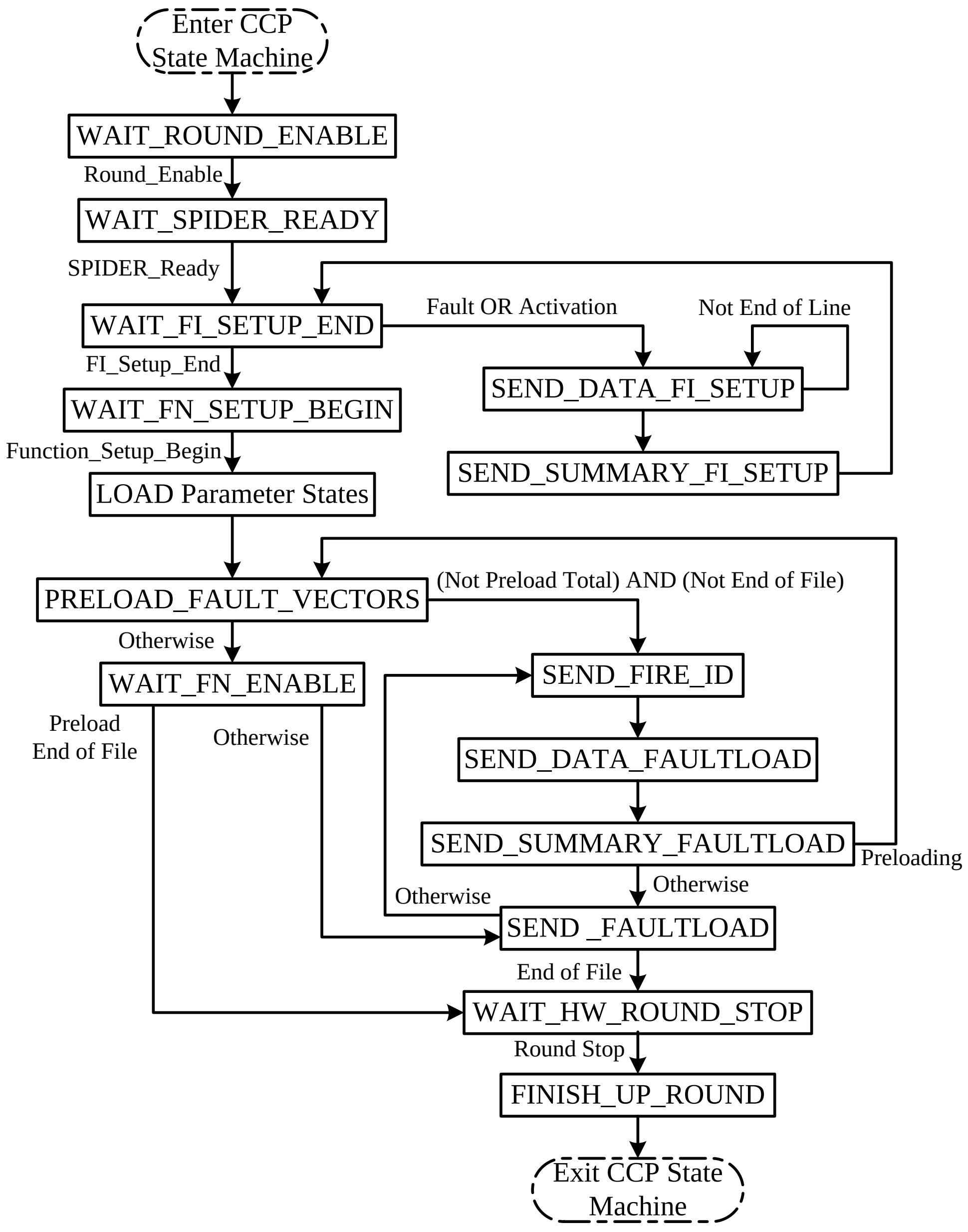


Figure 13.6: PTC Test Execution Software CCP State Machine

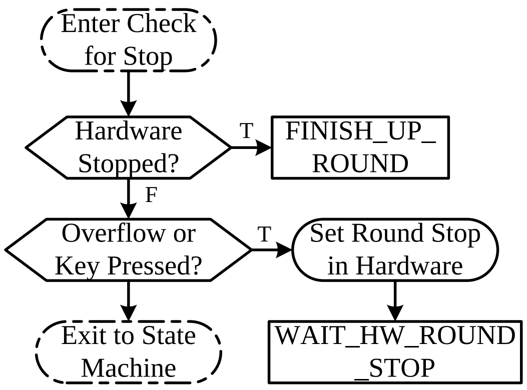


Figure 13.7: PTC and STC Test Execution Software Check for Stop Subset of CCP State Machine

Continuing in Figure 13.6, when none of the above stop conditions have occurred, the CCP state machine will continue normal operation of the states beginning with the WAIT_ROUND_ENABLE state. The software waits for the round to be enabled by the hardware in this state and then waits for the round controller to signal that SPIDER has completed initialization and is ready for operation in the WAIT_SPIDER_READY state. Then the software enters the WAIT_FI_SETUP_END state and the hardware is now in the fault injection setup mode. In this state, the Test Execution Software looks at the Test Specification data that was sent from the Data Management Software. The Test Specification file (see Appendix A) contains a number of setup lines. Each line can be one of the fault injection setup subsection. If there are no fault injections for the round, the No_FI line is found, the FI_Setup_End signal in the hardware is set, and there are no other fault injection setup subsection lines. Otherwise, the other fault injection setup subsection lines can be in any order with the FI_Setup_Done line at the end. If a fault or activation line is found in the Test Specification data, the number of data words to define the line may vary depending on the fault type. The state machine goes through a sequence of send states to transfer all of the necessary data to the hardware. When the FI_Setup_Done line is found, the FI_Setup_End signal is set in the hardware, allowing the state machine to begin waiting for the Function Setup round control mode.

When the hardware signals that it has entered the Function Setup mode by setting the Function_Setup_Begin signal during the WAIT_FN_SETUP_BEGIN state, the software CCP State Machine moves to a series of load states. In these states, the software interprets the Test Specification data and sends it to the hardware. The function setup data includes the Application_Assignment for the PEs, the number of ROBUS cycles per control cycle (RC_Per_CC), the number of control cycles per round (CC_Per_Round), and the PE communication schedule during a control cycle. The final two function setup load states send the sensor and command data to the hardware for the FMons to send during the round of execution. Also, while in the Function Setup mode, several fault vectors (see Section 4.3.2.2) are preloaded into the hardware when the round has fault injections. The CCP State Machine enters the PRELOAD_FAULT_VECTORS state, which will send the first part of the FI_FIRE TCM (see Section 7.3) as long as there is a fault vector that has not been sent in the Test Specification data and the total preloaded fault vectors to this point do not exceed a predetermined number. Then the CCP State Machine proceeds to a series of states that build and send the rest of the TCM to the hardware. Once the TCM is sent, the state machine returns to the PRELOAD_FAULT_VECTORS state. This cycle continues until either there are no more fault vectors in the round or the predetermined number of preloaded fault vectors is reached. In both cases, the Function_Setup_End flag is set signaling the hardware to start the Function Execution mode.

If the preloading ended because there were no more fault vectors and the end of file was reached, then the state machine transitions to the WAIT_HW_ROUND_STOP state. However, if there are more fault vectors, the state machine transitions to the SEND_FAULTLOAD state, where it sends the first part of the FI_FIRE TCM and then proceeds to the series of states that build and send the rest of the TCM to the hardware according to the format in Section 7.3. After the TCM is complete, the state machine transitions back to the SEND_FAULTLOAD state. As long as there are more fault vectors, the cycle from SEND_FAULTLOAD through the series of FI_FIRE TCM states continues. When there are no more fault vectors, which means the end of file was reached, then the CCP State Machine transitions to the WAIT_HW_ROUND_STOP state. At this point, the software is not required to do anything other than wait for the round to be completed by monitoring the Round_Stop signal from the hardware. The next state is FINISH_UP_ROUND, where the final stop condition is written to the Test Log data and the CCP_Round_Finished signal is set. The CCP State machine ends with this state. The CCP_Round_Finished triggers the exit of the main loop in the high level program of Figure 13.5 and sends the observation data to the Data Management Software before exiting the program.

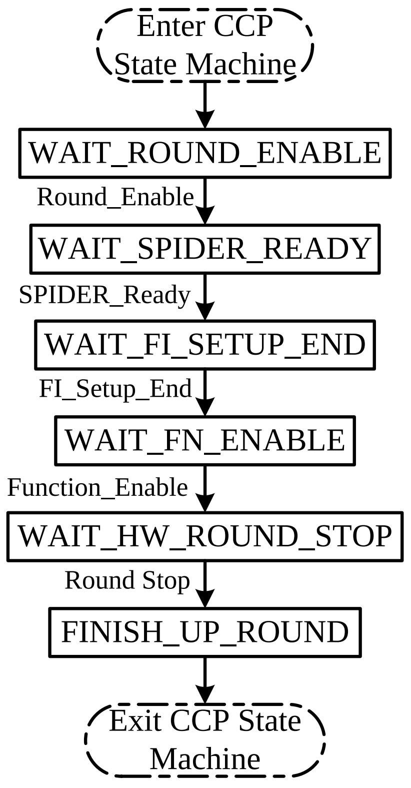


Figure 13.8: STC Test Execution Software CCP State Machine

The STC CCP State Machine tracks the hardware execution of a round closely but is not required to transfer as much data to the hardware as the PTC CCP State Machine. The STC CCP State Machine has a basic structure without loops or branching as shown in Figure 13.8. Similar to that of the PTC, the STC CCP State Machine checks for a stop condition in all states other than the FINISH_UP_ROUND state before executing the state. When there is no stop condition, the CCP State Machine continues execution. The first state is the WAIT_ROUND_ENABLE state. When the hardware signals that the round is enabled, the software gets the Round Index and the Enabled Nodes vector from the hardware that were specified by the PTC Test Execution Software in the initialization phase. Then the STC CCP State Machine transitions to the WAIT_SPIDER_READY state. Once the hardware round controller signals that SPIDER has completed initialization and is ready for operation, the state becomes the WAIT_FI_SETUP_END state. In the WAIT_FI_SETUP_END, WAIT_FN_ENABLE, and WAIT_HW_ROUND_STOP states, the CCP State Machine waits for the corresponding hardware signal to be set before continuing to the next state. The final state is the FINISH_UP_ROUND state which behaves the same as in the PTC CCP State Machine.

14. Final Remarks

The Configurable Fault-Injection and Monitoring System (CFIMS) is a flexible system that enables physical and simulated fault injection experiments for the systematic exploration of effects caused by adverse conditions on a distributed computation system running flight control applications. The CFIMS has a distributed architecture with capabilities for configurability and SUT functional stimulus generation, fault injection and state monitoring. This report has presented the CFIMS design concept and provided a comprehensive description of the first implementation.

The CFIMS has already been used in simulated fault injection experiments for an initial assessment of analytical performance models for a B747 flight control system under harsh conditions. The CFIMS was also used successfully in extensive physical fault injection experiments in a HIRF environment [Yates10]. Additional simulated fault injection experiments may be performed to study the behavior of the SUT under carefully controlled conditions in order to gain additional insight into the observed system response during the HIRF experiments and refine the analytical system models.

The CFIMS is currently being considered for fault injection experiments to support validation and verification activities for advanced distributed systems, such as an implementation of the self-stabilizing fault-tolerant clock synchronization protocol described in [Malekpour09] and a revised version of the ROBUS communication system with a robust self-stabilization capability.

Appendix A. Test Specification File

The Test Specification File contains the set of runtime parameters necessary for the CFIMS to perform a round of operation. The file is divided into the setup section and the execution section as shown in Figure A.1. The setup section of the file has all of the parameters needed before the start of the round. Each setup specification has one line and contains a text description, or label, stating the purpose of the line followed by zero or more values as shown in Figure A.2. The execution section of the file is located at the bottom and has one fault vector for each control cycle in a round. The fault vector lines do not have a text description at the beginning of each line.

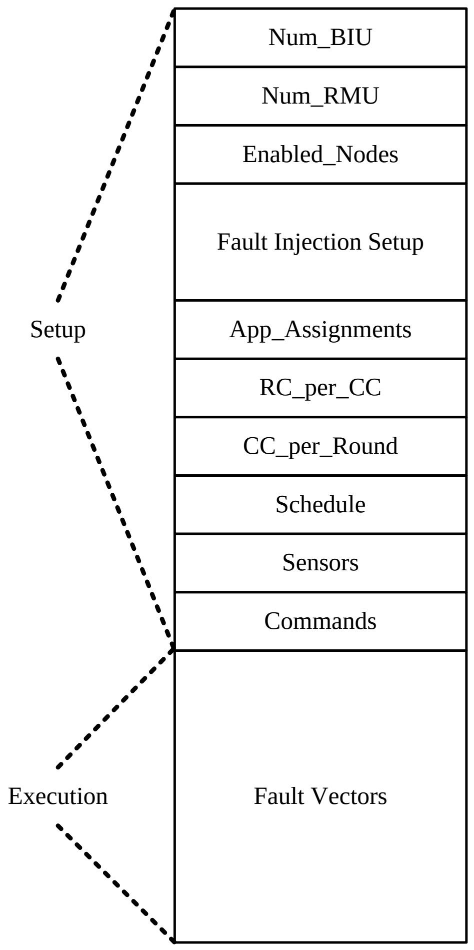


Figure A.1: Test Specification File Format

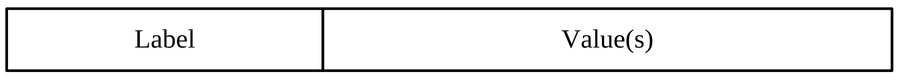


Figure A.2: Test Specification File Setup Line Format

Table A.1 gives a summary of all the types of lines in the setup section of the Test Specification file. The Num_BIU and Num_RMU lines in the setup section gives the software the SUT configuration used to generate the parameters for the hardware at synthesis. The Enabled_Nodes specification is a subset of the SUT configuration and indicates which PE-BIUs and RMUs will be active for the test. The Enabled_Nodes can handle up to twelve PE-BIUs and four RMUs (see Section 4.3 for the maximum configuration size) and contains one bit for each SPIDER node in the format given in Figure A.3. The fault injection setup subsection is located after the Enabled_Nodes line in the Test Specification file. This subsection is general because it defines the types of faults for the test. Fault and activation type lines that specify the behavior of the fault are within the fault injection setup section. Each specification has a 16-bit data word corresponding to each TCM Payload Word given in the FI_FAULT and FI_ACTIVATION TCMs (see Section 7.3). Since the fault injection capability is designed to be easily extended, the number of fault injection lines within the file varies, creating the need to denote the end of the fault injection setup section. The end of the section of the Test Specification file is indicated by the FI_Setup_Done line. When a test requires no fault injection to be done, a line with a text descriptor of NO_FI exists in place of the entire fault injection setup section of the file including the end of fault injection setup section line.

Table A.1: Test Specification File Setup Section Labels

Label	Description
Num_BIU	Number of BIUs in SUT configuration at synthesis
Num_RMU	Number of RMUs in SUT configuration at synthesis
Enabled_Nodes	Active PE-BIU and RMU nodes
No_FI	No fault injection during this test
RCI_Test_Vector	RCI within control cycle to activate the fault vectors
RCI_All_Disable	RCI within control cycle to deactivate the fault vectors
FI_Fault	TCM for fault type (see Section 4.3.2)
FI_Activation	TCM for activation pattern (see Section 4.3.2)
FI_Setup_Done	End of fault injection setup section
App_Assignments	PE application; Either IO communication or computing control commands
RC_per_CC	Number of ROBUS cycles per control cycle
CC_per_Round	Number of control cycles per round
Schedule	Number of data word transmission from each PE in configuration for the given RCI
Sensors	Sensor data words transmitted by the IO communication PEs
Commands	Command data words transmitted by the computing control commands PEs

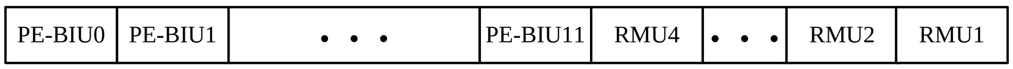


Figure A.3: Enabled_Nodes Format

The fault injection setup section of the Test Specification file either describes the fault injection to be completed or states that no fault injection is to be performed for the test. In the no fault injection case, the only line in the fault injection setup section is the line to specify that there is no fault injection. No extra data words or parameters are required in the no fault injection line. The line used to signal the end of the fault injection setup section is not needed when there is no fault injection because the section contains exactly one line. In Figure A.4, the TCM tag value for a fault type line is 1010 in binary, and the value for an activation type line is 1011 in binary. Figure A.4 has two fault and activation types (see Section 4.3.2) sent to specific Local Fault Injectors to show the generality of adding more types of faults in various locations within the SPIDER nodes. The first Fault Type 1 and Activation Type 2 are sent to the

Local Fault Injector with an ID of 1 inside PE-BIU 1. The second Fault Type 1 and Activation Type 3 are sent to the Local Fault Injector with an ID of 2 inside RMU 1. Note that the four lines that make up the two file descriptions can be in any order as long as they appear before the line designating the end of fault injection setup section.

FI_Fault	1010 0000 0100 0100	0000 0000 0000 0001
FI_Activation	1011 0000 0100 0100	0000 0000 0000 0010
FI_Fault	1010 0000 1000 0110	0000 0000 0000 0001
FI_Activation	1011 0000 1000 0110	0000 0000 0000 0011
FI_Setup_Done		

Figure A.4: Example of Fault Injection Section of Test Specification file with multiple fault types

Directly after the fault injection setup subsection is the App_Assignments where the applications are given to the PEs. The App_Assignments have the same format as the fault vectors except that the positions pertain to the enabled PEs. Each bit indicates if the application of the PE is an IO communication PE (0) or a PE computing the control commands (1). The RC_per_CC specification denotes the number of ROBUS cycles in one control cycle. The CC_per_Round line gives the number of control cycles to be executed during the round. Rather than setting the maximum number of control cycles, the CC_per_Round might be set to 0 to indicate the round will not end based on the number of control cycles. With the setting of 0, the test has to wait for another stop condition (see Section 5.2), such as executing all the given fault vectors, to end the round. The Schedule specification defines how many ROBUS messages (see Section 3.5) each PE in the configuration will send during the given ROBUS cycle. The first number of the Schedule specification is the RCI of each control cycle that has PEs transmitting with that schedule. Each subsequent number in the line is the number of messages each PE is scheduled to transmit for the RCI. Since a Schedule line only describes the PE transmissions for a single ROBUS cycle, multiple Schedule lines are required when PEs transmit on more than one ROBUS cycle per control cycle. The Sensors and Commands lines specify the data each PE transmits. Each number following the text description of these two lines is one data word. IO PEs send the sensor data, whereas Control PEs send the command data (see Section 4.1). The last section of the Test Specification file contains one fault vector (see Section 4.3.2) for each control cycle within the round. The format of the fault vector is the same as the format of the Enabled_Nodes given in Figure A.3.

Figure A.5 is an example Test Specification file with the appropriate line formats for the current version of the CFIMS. The system is configured to have 4 PE-BIUs and 3 RMUs. The Enabled_Nodes line specifies that PE 0, PE 1, PE 2, RMU 1, and RMU 2 are the only active nodes for this round. The fault injection setup section of this Test Specification file includes the RCI within the control cycle to begin and end the fault. The Fault Injection Controller activates the fault in RCI 0 and deactivates it in RCI 1. The fault and activation type lines are broadcast to all Local Fault Injectors. The only fault injectors in this version are located on the Reset signal of each node. Fault type 1 and activation type 2 are used in this setup to direct the Local Fault Injectors. Similar to the Figure A.4, the first four lines of the fault injection setup section can be in any order as long as they are before the end of fault injection setup line. The App_Assignments line makes PE 0 an IO PE and PEs 1 and 2 Control PEs. The example calls for 10 ROBUS cycles in each control cycle and does not set a maximum number of control cycles for the round. Each of the three active PEs is scheduled to transmit 50 words in RCI 1 as provided in the

Schedule line. The Sensors and Commands lines each give the 50 data word values that are transmitted by the PEs. The execution section contains a total of six fault vectors, which implies a total of six control cycles for the round. Most of the fault vectors are all zeros, indicating no fault injection for that control cycle. However, control cycles 3 and 5 have a fault in PE-BIU 2 and RMU 2, respectively.

```
Num_BIU             4
Num_RMU             3
Enabled_Nodes       1110000000000011
RCI_Test_Vector     0000000000000000
RCI_All_Disable     0000000000000001
FI_Fault            1010000000000001 0000000000000001
FI_Activation       1011000000000001 0000000000000010
FI_Setup_Done
App_Assignments     0110000000000000
RC_per_CC           10
CC_per_Round        0
Schedule            1 50 50 50 0
Sensors             7 100 87 98 7 53 17 37 62 52 …
Commands            65 82 77 58 36 80 2 62 48 77 …
0000000000000000
0000000000000000
0010000000000000
0000000000000000
0000000000000010
0000000000000000
```

Figure A.5: Test Specification File Example

Appendix B. Data Collected During Round Execution

This section describes the data collected by the CFIMS software processes during the execution of a round, at the end of which the data is organized into output files and sent to the repository for post-test analysis. The data collected at the PTC and STC covers every major system function, including round control, round time, function time, function monitoring, and state monitoring. As stated earlier in this report, the CFIMS can be viewed abstractly as providing a user service that reads a Test Specification and outputs a number of files containing the corresponding experimental data. From that perspective, this section is intended to provide a detailed description of the collected data at the CFIMS output service interface.

B.1. Round Control

The collected PTC and STC round control data are stored in corresponding separate log files with the following content.

1. A list of all Software Interface buffers that overflowed during the round, if any. Buffer overflows are an indication of an execution timing error and are generally accompanied by the loss of data. As such, they trigger an end to the execution of the round. At the PTC, the list of existing Software Interface buffers includes RTmr, FTmr, FMons, and SMons. At the STTC, the list includes RTmr and SMons.

2. A 16-bit status word as generated by the hardware Round Controller at the end of the round. The format is described in Table B.1. The bits are numbered 15 down to 0, with the leftmost bit numbered bit 15. See Section 5 for a description of the Controller Coordination Protocol (CCP) executed by the PTC and STC Round Controllers.

3. The round-stop condition is also saved as a separate item in decimal format.

Table B.1: Round Controller Status Word

Bit or Bit Slice	Variable	Description
15	Round_Enable	Set to 1 when the CCP Round Enable mode is complete. Otherwise, set to 0.
14	SPIDER_Ready	Set to 1 when the CCP SPIDER Initialization mode is complete. Otherwise, set to 0.
13	Function_Enable	Set to 1 when the CCP Function Execution mode begins. Otherwise, set to 0.
12	Execution_Stop	Set to 1 when the CCP System Stop mode begins. Otherwise, set to 0.
11	Round_Stop	Set to 1 when all hardware operation for the round stops. Otherwise, set to 0.
10	Stop_Echo_Timeout	Set to 1 by the CCP Stop Initiator when it does not receive an echo RC_STOP TCM. Otherwise, set to 0.
9-4	Stop_Condition	Condition that triggered the round stop. See Table B.2 for a full list of currently defined conditions.
3-0	---	Unused

Table B.2: Round Stop Conditions

Stop Condition Number	**Label**	**Description**
0	PTC_TCL_FAULT	An unexpected communication error was detected on a TCL link directly connected to the PTC (i.e., the CCL or a PTL). The error was detected by a PMCU receiver. Currently, this error reporting and round-stop triggering feature is not implemented.
1	PTC_RC_RCV_BUF_OVERFLOW	Buffer overflow on the CCL receive module of the PTC's Round Controller.
2	PTC_ENABLE_SEQ_ERROR	The PTC Round Controller received an unexpected RC_ENABLE TCM.
3	PTC_READY_SEQ_ERROR	The PTC Round Controller received an unexpected RC_READY TCM.
4	PTC_START_SEQ_ERROR	The PTC Round Controller received an unexpected RC_START TCM.
5	PTC_SPIDER_FAILURE	SPIDER failure detected by the SPIDER Health Monitor at the PTC.
6	PTC_SOFTWARE_STOP	Round stop triggered by the software processes at the PTC.
7	PTC_FI_STOP	Round stop triggered by the Fault Injection Controller at the PTC.
8	PTC_FUNCTION_STOP	Round stop triggered by the SPIDER Function Tester at the PTC.
9	PTC_STATE_MON_STOP	Round stop triggered by the PE-BIU State Monitor at the PTC.
10	PTC_ROUND_ENABLE_T_O	Timeout on the completion of the CCP Round Enable mode.
11	PTC_SPIDER_INIT_T_O	Timeout on the completion of the CCP SPIDER Initialization mode.
12	PTC_FI_SETUP_T_O	Timeout on the completion of the CCP Fault Injection Setup mode.
13	PTC_FUNCTION_SETUP_T_O	Timeout on the completion of the CCP Function Setup mode.
14	PTC_FUNCTION_DONE	Function execution completed at the PTC.
15	PTC_FI_DONE	Fault injection completed at the PTC.
16	STC_TCL_FAULT	An unexpected communication error was detected on a TCL link directly connected to the STC (i.e., the CCL or an STL). The error was detected by a PMCU receiver. Currently, this error reporting and round-stop triggering feature is not implemented.
17	STC_RC_RCV_BUF_OVERFLOW	Buffer overflow on the CCL receive module of the STC's Round Controller.
18	STC_ENABLE_SEQ_ERROR	The STC Round Controller received an unexpected RC_ENABLE TCM.
19	STC_READY_SEQ_ERROR	The STC Round Controller received an unexpected RC_READY TCM.
20	STC_START_SEQ_ERROR	The STC Round Controller received an unexpected RC_START TCM.
21	STC_SPIDER_FAILURE	SPIDER failure detected by the SPIDER Health Monitor at the STC.
22	STC_SOFTWARE_STOP	Round stop triggered by the software processes at the STC.
23	STC_FI_STOP	Round stop triggered by the Fault Injection Controller at the STC.

Stop Condition Number	Label	Description
24	STC_STATE_MON_STOP	Round stop triggered by the RMU State Monitor at the STC.
25	STC_ROUND_ENABLE_T_O	Timeout on the completion of the CCP Round Enable mode.
26	STC_SPIDER_INIT_T_O	Timeout on the completion of the CCP SPIDER Initialization mode.
27	STC_FI_SETUP_T_O	Timeout on the completion of the CCP Fault Injection Setup mode.
28	STC_FUNCTION_SETUP_T_O	Timeout on the completion of the CCP Function Setup mode.
29	STC_FI_DONE	Fault injection completed at the STC.

B.2. Round Timer

The PTC and STC Round Timers (RTmrs) serve as event timers for all the data records generated by the CFIMS. The RTmrs are frequently resynchronized to each other during the setup and execution phases of the round to enable the construction of a global timeline of events during the round. A new RTmr record is generated at the end of each RTmr interval, just before the Interval Time (IT) component is reset and the Interval Count (IC) is incremented by one. A record consists of (1 + SWXF_Num_RTmr_IC_DW) 16-bit data words. SWXF_Num_RTmr_IC_DW is a system configuration parameter currently set to 2. The first word is the value of the IT. The remaining words are 16-bit slices of the IC, with the most significant slice listed first.

B.3. Function Timer

An FTmr record is created when the FTime reaches the end of a ROBUS cycle or when the operational mode of the FTmr transitions from Preservation to Initialization. A record consists of (5 + (1 + SWXF_Num_RTmr_IC_DW)) 16-bit data words, with the first five words containing information about the FTmr state at the time the record was generated, and the remaining words are the record time tag with the format described for the RTmr in Section B.2. The format of the FTmr data is described in Table B.3.

Table B.3: Collected Function Timer Data

Word Number	Bit Slice	Bit Description Note: X_d denotes decimal value X.
1	Leftmost N bits	Accusations against PE(1..N). N is the total number of PEs. The leftmost bit corresponds to PE(1). The bit corresponding to PE(i) is set to 1 if the FTmr accused PE(i) in the time interval covered by the current record. Otherwise, the bit is set to 0.
	Rightmost 2 bits	FTmr mode: 0_d = Initialization, 1_d = Preservation.
2	Leftmost N bits	Conviction against PE(1..N). The leftmost bit corresponds to PE(1). The bit corresponding to PE(i) is set to 1 if the FTmr convicted PE(i) in the time interval covered by the current record. Otherwise, the bit is set to 0.
3	All 16 bits	ROBUS Time (RT)
4	All 16 bits	ROBUS Cycle Index (RCI)
5	All 16 bits	Control Cycle Index (CCI)

B.4. Function Monitors

The data records from all the Function Monitors (FMons) are saved to the same file. An FMon record consists of 1 + (SWXF_Num_CC_Obs_DW + (1 + SWXF_Num_RTmr_IC_DW)) data items, with the first item being a decimal-valued FMon identifier with a unique value assignment for each of the N FMons.

The next SWXF_Num_CC_Obs_DW (currently set to 2) 16-bit words contain the observations made by the FMon based on the received transmissions over the PTL from its corresponding PE during the control cycle. An FMon outputs N observations, one per PE with each PE viewed as a source of messages on ROBUS, independent of the actual communication schedule. The observations are packaged into a long bit string formed of consecutive end-to-end bit slices starting with the observations for PE 1 at the leftmost position. Each observation bit-slice has a width of CC_Obs_Width bits (currently set to 4) and is binary encoded as shown in Table B.4 (using decimal format). (A full description of the FMon observations can be found in Section 4.3.1.2.) The full observation bit string is divided into SWXF_Num_CC_Obs_DW 16-bit slices when the record is created.

The remaining words in an FMon record are the time tag with the format described for the RTmr in Section B.2.

Table B.4: Observation codes for a Function Monitor

Observation Code (in decimal format)	**Observation by FMon$_j$**
0	Omitted Sender Id
1	Invalid Sender Id
2	Repeated Sender Id
3	Bad Payload Length
4	Detected Reception Error at Receiver PE
5	Detected Reception Error at Sender PE
6	Bad Message Content
7	Good Message

B.5. PE-BIU State Monitors

The data records from all the lane State Monitors (SMons) are saved to the same file. An SMon record consists of 1 + (PE_BIU_State_Msg_Length + 1 + (1 + SWXF_Num_RTmr_IC_DW)) data items, with the first item being a decimal-valued SMon lane identifier with a unique value assignment for each of the N lanes. Note that the number of PEs (denoted Num_PE) is equal to the number of BIUs (denoted Num_BIU). Both of these quantities are also denoted N.

The content of the next PE_BIU_State_Msg_Length + 1 (currently, a total of 9) 16-bit words is described in Table B.5. The words in the state message are updated during normal operation as indicated in the Table, or immediately at the time of an unscheduled-message triggering event (i.e., an RPP failure or a PE failure).

The remaining words in an SMon lane record are the time tag with the format described for the RTmr in Section B.2.

Table B.5: Main state record content for a PE-BIU State Monitor Lane

Word Number	Word Update Condition	Bit or Bit Slice	Bit Description
1	This word is updated every time a new message is generated.	Leftmost TCM_Tag_Width (currently 4) bits	Binary encoded value of the TCM tag as set by the Embedded Node Monitor at the PE-BIU node. This corresponds to tag SM_PE_BIU.
		Rightmost State_Msg_Seq_Num_Width (currently 8) bits	State message sequence number. The sequence number is incremented by one every time a new state message is generated, and it rolls over after reaching the maximum binary value.
2	This word is updated every time a new message is generated.	Leftmost Node_Monitor_RTS_Trigger_Width (currently 4) bits	Binary encoded value of the condition that triggered the state message. The currently defined values are as follows. 0 = RPP_FAILURE 1 = PE_FAILURE 2 = STM_BEGIN 3 = CDM_BEGIN 4 = CIM_BEGIN 5 = CJM_SCHED_UP 6 = CPM_SCHED_UP 7 = TCL_ERROR (currently not used)
		Rightmost Num_RMU bits	Status of the BIU's word-mode communication units since the last data record was generated. The rightmost bit is the status for the first communication unit (i.e., the one receiving from RMU 1); the second rightmost bit is the status for the second communication unit; etc. A communication unit's status bit is set to 0 if the unit reported an invalid input waveform at any time since the last state message was triggered. A value of 1 indicates that the corresponding communication unit consistently detected a valid Manchester-encoded input signal since the last state message was triggered.
3	This word is updated whenever the RPP/BIU Mode Control Unit (MCU) issues a command satisfying one of the following	Bit 15 (Leftmost bit)	RPP Input Unit's (IU) Zero_Schedule flag; 0 = non-empty schedule 1 = new schedule is 0 (i.e., empty; no scheduled PE messages in the current ROBUS cycle)

Word Number	Word Update Condition	Bit or Bit Slice	Bit Description
	conditions: the first command in Self-Test mode, the first in Clique Detection Mode, the first in Clique Initialization mode, a Schedule Update command in Clique Join mode, or a Schedule Update command in Clique Preservation mode.	14	RPP Input Unit's (IU) Invalid_Schedule flag; 0 = new schedule is valid 1 = new schedule is invalid
		13	Node kind: 0 = PE_BIU, 1 = RMU
		12-9 (Next RPP_Node_Id_Vect_Width (currently 4) bits)	RPP Node Id with valid binary value range 1 to max(Num_RMU, Num_BIU).
		8-6 (Next RPP_Major_Mode_Vect_Width (currently 3) bits)	Binary encoded value of the RPP major mode; 0 = Self Test 1 = Clique Detection 2 = Clique Join 3 = Clique Initialization 4 = Clique Preservation
		5	Current diagnostic cycle when the current major mode is Clique Join mode; 0 = diagnostic cycle 0 1 = diagnostic cycle 1
		4-2 (Next RPP_Minor_Mode_Vect_Width (currently 3) bits)	Binary encoded value of the RPP minor mode; 0 = Reset 1 = Self Test 2 = Preliminary Diagnosis and Synchronization Capture 3 = Initial Diagnosis and Initial Synchronization 4 = Collective Diagnosis 5 = Schedule Update 6 = PE Communication 7 = Synchronization Preservation
		1-0 (Next RPP_Sched_Status_Vect_Width (currently 2) bits)	Binary encoded value of the schedule status reported by the MCU; 0 = Valid 1 = Zero 2 = Invalid
4	This word is updated whenever the RPP/BIU enters the Collective Diagnosis minor mode.	Leftmost Num_RMU bits	Accusations against RMUs generated by the local RPP/BIU. The leftmost bit corresponds to RMU 1. 0 = not accused 1 = accused

Word Number	Word Update Condition	Bit or Bit Slice	Bit Description
		Rightmost Num_BIU bits	Accusations against BIUs in reversed order. The rightmost bit corresponds to BIU 1. 0 = not accused 1 = accused
5	This word is updated when the RPP/BIU enters the Schedule Update minor mode in the Clique Join or Clique Preservation major mode.	Leftmost Num_RMU bits	Convictions against RMUs generated by the local RPP/BIU. The leftmost bit corresponds to RMU 1. 0 = not convicted 1 = convicted
		Rightmost Num_BIU bits	Convictions against BIUs in reversed order. The rightmost bit corresponds to BIU 1. 0 = not convicted 1 = convicted
6	This word is updated when there is an RPP detected failure.	Bit 15 (leftmost bit)	SMU_Failure flag generated by the RPP Status Monitoring Unit (SMU); 0 = no detected failure 1 = detected failure
		14	RPP's SMU_No_Clique flag; 0 = no-clique condition not detected 1 = no-clique condition detected
		13	RPP's SMU_Accusation_Against_Self flag; 0 = RPP has not accused itself 1 = RPP has accused itself
		12	RPP's SMU_OK_Distrusted flag; 0 = RPP trusts at least one node of the opposite kind (i.e., RMUs) 1 = RPP distrusts (i.e., has accused or convicted) every node of the opposite kind
		11	RPP's SMU_SK_Distrusted flag; 0 = RPP trusts at least one node of the same kind (i.e., BIUs) 1 = RPP distrusts (i.e., has accused or convicted) every node of the same kind

Word Number	Word Update Condition	Bit or Bit Slice	Bit Description
		10	RPP's SMU_Timeout flag; 0 = no timeout detected 1 = SMU detected a timeout condition for Synchronization Capture in the Clique Detection mode, Initial Synchronization in the Clique Initialization mode, or for a ROBUS cycle in the Clique Join or Clique Preservation modes.
		9	RPP's SMU_Protocol_Error flag; 0 = no protocol error detected 1 = protocol error detected
		8-4 (Next RPP_SMU_Protocol_Error_Code_Width (currently 5) bits)	Binary encoded value of RPP's MU_Protocol_Error_Code signal. See Appendix B in [Torres08B] for a full list of error codes.
7	This word is updated when there is a detected PE failure or the RPP issues a Schedule Update command in the Clique Preservation mode.	Bits 15-14 (Leftmost PE_OCL_Width (currently 2) bits)	Binary encoded value of the PE major mode (OCL = Operation Coordination Level); 0 = OCL0 1 = OCL1 2 = OCL2 3 = OCL3
		13	PE_Failure flag; 0 = PE did not detect a failure condition 1 = PE detected a failure condition
		12-8 (Next PE_Failure_Code_Width (currently 5) bits)	Binary encoded value of the failure code reported by the PE. See the VHDL description of the PE Master Controller's finite state machine for the conditions corresponding to the reported failure codes.
8	This word is updated when there is a detected PE failure or the RPP issues a Schedule Update command in the Clique Preservation mode.	All 16 bits	ROBUS Cycle Index (RCI) as reported by the PE.

Word Number	Word Update Condition	Bit or Bit Slice	Bit Description
9	This word is generated at the PTC.	Bits 15-14 (Leftmost Node_Condition_Width (currently 2) bits)	Binary encoded value of the node condition as reported by the Health Monitor at the lane State Monitor for the corresponding PE-BIU node. 0 = Disabled 1 = Recovering 2 = Restored 3 = unused code

B.6. RMU State Monitors

The data records generated by the RMU State Monitor have essentially the same format at the PE-BIU records. The main difference is due to the absence of PEs at the RMU nodes.

The data records from all the lane State Monitors (SMons) are saved to the same file. An SMon record consists of 1 + (RMU_State_Msg_Length + 1 + (1 + SWXF_Num_RTmr_IC_DW)) data items, with the first item being a decimal-valued SMon lane identifier with a unique value assignment for each of the SMon lanes monitoring the RMUs. The number of RMUs is denoted Num_RMU.

Note that the number of PEs (denoted Num_PE) is equal to the number of BIUs (denoted Num_BIU). Both of these quantities are also denoted N.

The content of the next RMU_State_Msg_Length + 1 (currently, a total of 7) 16-bit words is described in Table B.6. The words in the state message are updated during normal operation as indicated in the Table, or immediately at the time of an unscheduled-message triggering event (i.e., an RPP failure).

The remaining words in an SMon lane record are the time tag with the format described for the RTmr in Section B.2.

Table B.6: Main state record content for an RMU State Monitor Lane

Word Number	Word Update Condition	Bit or Bit Slice	Bit Description
1	This word is updated every time a new message is generated.	Leftmost TCM_Tag_Width (currently 4) bits	Binary encoded value of the TCM tag as set by the Embedded Node Monitor at the RMU node. This should correspond to tag SM_RMU.
		Rightmost State_Msg_Seq_Num_Width (currently 8) bits	State message sequence number. The sequence number is incremented by one every time a new state message is generated, and it rolls over after reaching the maximum binary value.
2	This word is updated every time a new message is generated.	Leftmost Node_Monitor_RTS_Trigger_Width (currently 4) bits	Binary encoded value of the condition that triggered the state message. The currently defined values are as follows. 0 = RPP_FAILURE 1 = PE_FAILURE (not applicable to RMU nodes) 2 = STM_BEGIN 3 = CDM_BEGIN 4 = CIM_BEGIN 5 = CJM_SCHED_UP 6 = CPM_SCHED_UP 7 = TCL_ERROR (not currently used)
		Rightmost Num_BIU bits	Status of the RMU's word-mode communication units since the last data record was generated. The rightmost bit is the status for the first communication unit (i.e., the one receiving from BIU 1); the second rightmost bit is the status for the second communication unit; etc. A communication unit bit is set to 0 if the unit reported an invalid input waveform at any time since the last state message was triggered. A value of 1 indicates that the corresponding communication unit consistently detected a valid Manchester-encoded input signal since the last state message was triggered.
3	This word is updated whenever the RPP/RMU Mode Control Unit (MCU) issues a command satisfying the following	Bit 15 (Leftmost bit)	RPP Input Unit's (IU) Zero_Schedule flag; 0 = non-empty schedule 1 = new schedule is 0 (i.e., empty; no scheduled PE messages during the current ROBUS cycle)

Word Number	Word Update Condition	Bit or Bit Slice	Bit Description
	conditions: the first command in Self-Test mode, the first in Clique Detection Mode, the first in Clique Initialization mode, a Schedule Update command in Clique Join mode, or a Schedule Update command in Clique Preservation mode.	14	RPP Input Unit's (IU) Invalid_Schedule flag; 0 = new schedule is valid 1 = new schedule is invalid
		13	Node kind: 0 = PE_BIU, 1 = RMU
		12-9 (Next RPP_Node_Id_Vect_Width (currently 4) bits)	RPP Node Id with binary value range 1 to Max_Value(Num_RMU, Num_BIU)
		8-6 (Next RPP_Major_Mode_Vect_Width (currently 3) bits)	Binary encoded value of the RPP major mode; 0 = Self Test 1 = Clique Detection 2 = Clique Join 3 = Clique Initialization 4 = Preservation
		5	Current diagnostic cycle when the current major mode is Clique Join mode; 0 = diagnostic cycle 0 1 = diagnostic cycle 1
		4-2 (Next RPP_Minor_Mode_Vect_Width (currently 3) bits)	Binary encoded value of the RPP minor mode; 0 = Reset 1 = Self Test 2 = Preliminary Diagnosis and Synchronization Capture 3 = Initial Diagnosis and Initial Synchronization 4 = Collective Diagnosis 5 = Schedule Update 6 = PE Communication 7 = Synchronization Preservation
		1-0 (Next RPP_Sched_Status_Vect_Width (currently 2) bits)	Binary encoded value of the schedule status reported by the MCU; 0 = Valid 1 = Zero 2 = Invalid
4	This word is updated whenever the RPP/RMU enters the Collective Diagnosis minor mode.	Leftmost Num_BIU bits	Accusations against BIUs generated by the local RPP/RMU. The leftmost bit corresponds to BIU 1. 0 = not accused 1 = accused

Word Number	Word Update Condition	Bit or Bit Slice	Bit Description
		Rightmost Num_RMU bits	Accusations against RMUs in reversed order. Rightmost bit corresponds to RMU 1. 0 = not accused 1 = accused
5	This word is updated when the RPP/RMU enters the Schedule Update minor mode in the Clique Join or Clique Preservation major modes.	Leftmost Num_BIU bits	Convictions against BIUs generated by the local RPP/RMU. The leftmost bit corresponds to BIU 1. 0 = not convicted 1 = convicted
		Rightmost Num_RMU bits	Convictions against RMUs in reversed order. Rightmost bit corresponds to RMU 1. 0 = not convicted 1 = convicted
6	This word is updated when there is an RPP detected failure.	Bit 15 (leftmost bit)	Value of the SMU_Failure flag generated by the RPP Status Monitoring Unit (SMU); 0 = no detected failure 1 = detected failure
		14	RPP’s SMU_No_Clique flag; 0 = no-clique condition not detected 1 = no-clique condition detected
		13	RPP’s SMU_Accusation_Against_Self flag; 0 = RPP has not accused itself 1 = RPP has accused itself
		12	RPP’s SMU_OK_Distrusted flag; 0 = RPP trusts at least one node of the opposite kind (i.e., BIUs) 1 = RPP distrusts (i.e., has accused or convicted) every node of the opposite kind
		11	RPP’s SMU_SK_Distrusted flag; 0 = RPP trusts at least one node of the same kind (i.e., RMUs) 1 = RPP distrusts (i.e., has accused or convicted) every node of the same kind

Word Number	Word Update Condition	Bit or Bit Slice	Bit Description
		10	RPP's SMU_Timeout flag; 0 = no timeout detected 1 = SMU detected a timeout condition for Synchronization Capture in the Clique Detection mode, Initial Synchronization in the Clique Initialization mode, or for a ROBUS cycle in the Clique Join or Clique Preservation modes.
		9	RPP's SMU_Protocol_Error flag; 0 = no protocol error detected 1 = protocol error detected
		8-4 (Next RPP_SMU_Protocol_Error_Code_Width (currently 5) bits)	Binary encoded value of RPP's SMU_Protocol_Error_Code signal. See Appendix B in [Torres08B] for a full list of error codes.
7	This word is generated at the STC.	Bits 15-4 (Leftmost Node_Condition_Width (currently 2) bits)	Binary encoded value of the node condition as reported by the Health Monitor at the lane State Monitor for the corresponding RMU node. 0 = Disabled 1 = Recovering 2 = Restored 3 = unused code

Appendix C. Test Control Software Pseudo-Code

The Test Control Software of the CFIMS is composed of Data Management and Test Execution Software for each Test Controller (PTC and STC) as described in Section 13. The following sections give the detailed pseudo-code of these software programs.

C.1. PTC Data Management Software

```
Open Test_Spec file
Count number of characters in specification part of Test_Spec and save the characters and count in RAM
Count fault vectors from Test_Spec and save the fault vectors and count in RAM
Close Test_Spec file
Create a socket
Get Host Name IP address
Bind the socket to the IP address or Name

Listen for connection request
Enable standard I/O on socket

///////////////////////////////////////////////////////////////////////////////////////
//Send Test spec character section
Make all of the following writes blocking
Status = Send_UInt_To_Socket(TS_char_count)
If errno != 0
{
    Display on screen "Error in sending Test Spec count: (errno)"
    Exit Program
}
Else If status != 4
{
    Display on screen "Write returned status of (status) on Test Spec count"
    Exit Program
}

For each character in specification
    Send_Char_To_Socket(TS_char)

//Check that the correct number of Test spec characters was received at client
Make all of the following reads non-blocking
Start timer 1 – Client Test Spec character count
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Cli_TS_char_count)
    If (status == -1) and (timer >= TIMEOUT_CLI_TS_CHAR_CNT)
    {
        Display on screen "Timeout occurred while waiting for Test Spec character count"
        Exit Program
    }
}
If status = 0
{
    Display on screen "Socket was closed prematurely when trying to read Test Spec character count"
    Exit Program
```

```
}
Else If status < sizeof(Cli_TS_char_count)
{
    Display on screen "Partial Test Spec character count received: (status) bytes"
    Exit Program
}
Else If Cli_TS_char_count != TS_char_count //check that client got all
{
    Display on screen "Client received (Cli_TS_char_count) Test Spec characters when (TS_char_count) were
        sent"
    Exit Program
}
//Else it was successful

////////////////////////////////////////////////////////////////////////////////
//Send fault vectors
Make all of the following writes blocking
Status = Send_UInt_To_Socket(FV_count)
If errno != 0
{
    Display on screen "Error in sending fault vector count: (errno)"
    Exit Program
}
Else If status != 4
{
    Display on screen "Write returned status of (status) on fault vector count"
    Exit Program
}

For each fault vector //if there are none, it will go on
{
    Send_USInt_To_Socket(fault_vector[])
    If errno != 0
    {
        Display on screen "Error in sending fault vector data: (errno)"
        Write_All_Output_To_File()
        Exit Program
    }
    Else If status != 2
    {
        Display on screen "Write returned status of (status) on fault vector data"
        Write_All_Output_To_File()
        Exit Program
    }
}

//Check that the correct number of fault vectors was received at client
Make all of the following reads non-blocking
Start timer 2 – Client fault vectors count
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Cli_FV_count)
    If (status == -1) and (timer >= TIMEOUT_CLI_FV_CNT)
    {
            Display on screen “Timeout occurred while waiting for fault vector count”
            Exit Program
        }
}
If status = 0
{
        Display on screen “Socket was closed prematurely when trying to read fault vector count”
        Exit Program
}
Else If status < sizeof(Cli_FV_count)
{
        Display on screen “Partial fault vector count received: (status) bytes”
        Exit Program
}
Else If Cli_FV_count!= FV_count //check that client got all
{
        Display on screen “Client received (Cli_FV_count) fault vector entries when (FV_count) were sent”
        Exit Program
}
//Else it was successful
```

///
//Open files
Open 5 output files – 4 binary (FMon, FTmr, RTmr, SMon) and 1 text (Test_Log)

///
//Receive Test_Log data and write to file

```
//Not have it timeout #3
Initialize status = -1
While status = -1
{
        Status = Read_UInt_From_Socket(Cli_TL_count) //This one will wait until there is data
        If (status == -1) and (Key_Pressed)
        {
            Display on screen “Operator pressed a key while waiting for Test Log character count to end the program”
            Exit Program
        }
}
If status = 0
{
        Display on screen “Socket was closed prematurely when trying to read Test Log character count”
        Exit Program
}
Else If status < sizeof(Cli_TL_count)
{
        Display on screen “Partial Test Log character count received: (status) bytes”
        Exit Program
}

Start timer 4 – Test Log characters
TL_count = 0
While (TL_count < Cli_TL_count)
{
        Initialize status = -1
        While status = -1
```

```
    {
        Status = Read_Char_From_Socket(TL_char)
        If (status == -1) and (timer >= TIMEOUT_DATA_WORD)
        {
            Display on screen "Timeout occurred while waiting for Test Log characters"
            Exit Program
        }
    }
    If status = 0
    {
        Display on screen "Socket was closed prematurely when trying to read Test Log characters"
        Exit Program
    }
    Else If status < sizeof(TL_char)
    {
        Display on screen "Partial Test Log characters received: (status) bytes"
        Exit Program
    }
    //Else it was successful
    TL_count++
    Write_char(Test_Log_File, TL_char)
    Reset timer
}

//Send count of characters received for check
Make all of the following writes blocking
Status = Send_UInt_To_Socket(TL _count)
If errno != 0
{
    Display on screen "Error in sending test log count: (errno)"
    Exit Program
}
Else If status != 4
{
    Display on screen "Write returned status of (status) on test log count"
    Exit Program
}

////////////////////////////////////////////////////////////////////////////////
//Receive FMon data and write to file
Make all of the following reads non-blocking
Start timer 5 – FMon count
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Cli_FMon_count)
    If (status == -1) and (timer >= TIMEOUT_FMON_CNT)
    {
        Display on screen "Timeout occurred while waiting for FMon count"
        Exit Program
    }
}
If status = 0
{
    Display on screen "Socket was closed prematurely when trying to read FMon count"
    Exit Program
}
Else If status < sizeof(Cli_FMon_count)
{
    Display on screen "Partial FMon count received: (status) bytes"
    Exit Program
}

Start timer 6 – FMon
FMon_count = 0
While (FMon_count < Cli_FMon_count)
{
    Initialize status = -1
    While status = -1
    {
        Status = Read_USInt_From_Socket(FMon_val)
        If (status == -1) and (timer >= TIMEOUT_DATA_WORD)
        {
            Display on screen "Timeout occurred while waiting for FMon [FMon_count]"
            Exit Program
        }
    }
    If status = 0
    {
        Display on screen "Socket was closed prematurely when trying to read FMon [FMon_count]"
        Exit Program
    }
    Else If status < sizeof(FMon_val)
    {
        Display on screen "Partial FMon received: (status) bytes"
        Exit Program
    }
    //Else it was successful
    FMon[FMon_count] = FMon_val
    FMon_count++
    Reset timer
}

Write each FMon[] value to binary FMon_File

//Send count of entries received for check
Make all of the following writes blocking
Status = Send_UInt_To_Socket(FMon_count)
If errno != 0
{
    Display on screen "Error in sending FMon count: (errno)"
    Write_All_Output_To_File()
    Exit Program
}
Else If status != 4
{
    Display on screen "Write returned status of (status) on FMon count"
    Write_All_Output_To_File()
    Exit Program
}
```

```
///////////////////////////////////////////////////////////////////////////////
//Receive FTmr data and write to file
Make all of the following reads non-blocking
Start timer 7 – FTmr count
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Cli_ FTmr _count)
    If (status == -1) and (timer >= TIMEOUT_FTMR_CNT)
    {
        Display on screen “Timeout occurred while waiting for FTmr count”
        Exit Program
    }
}
If status = 0
{
    Display on screen “Socket was closed prematurely when trying to read FTmr count”
    Exit Program
}
Else If status < sizeof(Cli_FTmr_count)
{
    Display on screen “Partial FTmr count received: (status) bytes”
    Exit Program
}

Start timer 8 – FTmr
FTmr_count = 0
While (FTmr_count < Cli_FTmr_count)
{
    Initialize status = -1
    While status = -1
    {
        Status = Read_USInt_From_Socket(FTmr_val)
        If (status == -1) and (timer >= TIMEOUT_DATA_WORD)
        {
            Display on screen “Timeout occurred while waiting for FTmr [FTmr_count]”
            Exit Program
        }
    }
    If status = 0
    {
        Display on screen “Socket was closed prematurely when trying to read FTmr [FTmr_count]”
        Exit Program
    }
    Else If status < sizeof(FTmr_val)
    {
        Display on screen “Partial FTmr received: (status) bytes”
        Exit Program
    }
    //Else it was successful
    FTmr_count++
    Write FTmr_val to binary file FTmr_File
    Reset timer
}
```

```
//Send count of entries received for check
Make all of the following writes blocking
Status = Send_UInt_To_Socket(FTmr_count)
If errno != 0
{
    Display on screen “Error in sending FTmr count: (errno)”
    Write_All_Output_To_File()
    Exit Program
}
Else If status != 4
{
    Display on screen “Write returned status of (status) on FTmr count”
    Write_All_Output_To_File()
    Exit Program
}
```

//

//Receive RTmr data and write to file

```
Make all of the following reads non-blocking
Start timer 9 – RTmr count
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Cli_ RTmr _count)
    If (status == -1) and (timer >= TIMEOUT_RTMR_CNT)
    {
        Display on screen “Timeout occurred while waiting for RTmr count”
        Exit Program
    }
}
If status = 0
{
    Display on screen “Socket was closed prematurely when trying to read RTmr count”
    Exit Program
}
Else If status < sizeof(Cli_ RTmr _count)
{
    Display on screen “Partial RTmr count received: (status) bytes”
    Exit Program
}

Start timer 10 – RTmr
RTmr_count = 0
While (RTmr_count < Cli_RTmr_count)
{
    Initialize status = -1
    While status = -1
    {
        Status = Read_USInt_From_Socket(RTmr_val)
        If (status == -1) and (timer >= TIMEOUT_DATA_WORD)
        {
            Display on screen “Timeout occurred while waiting for RTmr [RTmr_count]”
            Exit Program
        }
    }
    If status = 0
    {
        Display on screen “Socket was closed prematurely when trying to read RTmr [RTmr_count]”
        Exit Program
    }
    Else If status < sizeof(RTmr_val)
    {
        Display on screen “Partial RTmr received: (status) bytes”
        Exit Program
    }
    //Else it was successful
    RTmr_count++
    Write RTmr_val to binary file RTmr_File
    Reset timer
}

//Send count of entries received for check
Make all of the following writes blocking
Status = Send_UInt_To_Socket(RTmr_count)
If errno != 0
{
    Display on screen “Error in sending RTmr count: (errno)”
    Write_All_Output_To_File()
    Exit Program
}
Else If status != 4
{
    Display on screen “Write returned status of (status) on RTmr count”
    Write_All_Output_To_File()
    Exit Program
}

/////////////////////////////////////////////////////////////////////////////////
//Receive SMon data and write to file
Make all of the following reads non-blocking
Start timer 11 – SMon count
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Cli_SMon_count)
    If (status == -1) and (timer >= TIMEOUT_SMON_CNT)
    {
        Display on screen “Timeout occurred while waiting for SMon count”
        Exit Program
    }
}
If status = 0
{
    Display on screen “Socket was closed prematurely when trying to read SMon count”
    Exit Program
}
Else If status < sizeof(Cli_SMon_count)
{
    Display on screen “Partial SMon count received: (status) bytes”
    Exit Program
}

Start timer 10 – SMon
SMon_count = 0
While (SMon_count < Cli_SMon_count)
{
    Initialize status = -1
    While status = -1
    {
        Status = Read_USInt_From_Socket(SMon_val)
        If (status == -1) and (timer >= TIMEOUT_DATA_WORD)
        {
            Display on screen "Timeout occurred while waiting for SMon [SMon_count]"
            Exit Program
        }
    }
    If status = 0
    {
        Display on screen "Socket was closed prematurely when trying to read SMon [SMon_count]"
        Exit Program
    }
    Else If status < sizeof(SMon_val)
    {
        Display on screen "Partial SMon received: (status) bytes"
        Exit Program
    }
    //Else it was successful
    SMon_count++
    Write SMon_val to binary file SMon_File
    Reset timer
}

//Send count of entries received for check
Make all of the following writes blocking
Status = Send_UInt_To_Socket(SMon_count)
If errno != 0
{
    Display on screen "Error in sending SMon count: (errno)"
    Write_All_Output_To_File()
    Exit Program
}
Else If status != 4
{
    Display on screen "Write returned status of (status) on SMon count"
    Write_All_Output_To_File()
    Exit Program
}

Close files and socket
```

C.2. STC Data Management Software

```
Create a socket
Get Host Name IP address
Bind the socket to the IP address or Name

Listen for connection request
Create child process to service the client
Enable standard I/O on socket
Make all of the following reads non-blocking

Open 3 output files – 2 binary (RTmr, SMon) and 1 text (Test_Log)

//////////////////////////////////////////////////////////////////////////////////
//Receive Test_Log data and write to file
//Not have it timeout #1
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Cli_TL_count) //This one will wait until there is data
    If (status == -1) and (Key_Pressed)
    {
        Display on screen "Operator pressed a key while waiting for Test Log character count to end the program"
        Exit Program
    }
}
If status = 0
{
    Display on screen "Socket was closed prematurely when trying to read Test Log character count"
    Exit Program
}
Else If status < sizeof(Cli_TL_count)
{
    Display on screen "Partial Test Log character count received: (status) bytes"
    Exit Program
}

Start timer 2 – Test Log characters
TL_count = 0
While (TL_count < Cli_TL_count)
{
    Initialize status = -1
    While status = -1
    {
        Status = Read_Char_From_Socket(TL_char)
        If (status == -1) and (timer >= TIMEOUT_DATA_WORD)
        {
            Display on screen "Timeout occurred while waiting for Test Log characters"
            Exit Program
        }
    }
    If status = 0
    {
        Display on screen "Socket was closed prematurely when trying to read Test Log characters"
        Exit Program
    }
    Else If status < sizeof(TL_char)
    {
        Display on screen "Partial Test Log characters received: (status) bytes"
        Exit Program
    }
    //Else it was successful
    TL_count++
    Write_char(Test_Log_File, TL_char)
    Reset timer
}

//Send count of characters received for check
Make all of the following writes blocking
Status = Send_UInt_To_Socket(TL _count)
If errno != 0
{
    Display on screen "Error in sending test log count: (errno)"
    Exit Program
}
Else If status != 4
{
    Display on screen "Write returned status of (status) on test log count"
    Exit Program
}

/////////////////////////////////////////////////////////////////////////////////
//Receive RTmr data and write to file
Make all of the following reads non-blocking
Start timer 3 – RTmr count
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Cli_ RTmr _count)
    If (status == -1) and (timer >= TIMEOUT_RTMR_CNT)
    {
        Display on screen "Timeout occurred while waiting for RTmr count"
        Exit Program
    }
}
If status = 0
{
    Display on screen "Socket was closed prematurely when trying to read RTmr count"
    Exit Program
}
Else If status < sizeof(Cli_ RTmr _count)
{
    Display on screen "Partial RTmr count received: (status) bytes"
    Exit Program
}

Start timer 4 – RTmr
RTmr_count = 0
While (RTmr_count < Cli_RTmr_count)
{
        Initialize status = -1
        While status = -1
        {
                Status = Read_USInt_From_Socket(RTmr_val)
                If (status == -1) and (timer >= TIMEOUT_DATA_WORD)
                {
                        Display on screen “Timeout occurred while waiting for RTmr [RTmr_count]”
                        Exit Program
                }
        }
        If status = 0
        {
                Display on screen “Socket was closed prematurely when trying to read RTmr [RTmr_count]”
                Exit Program
        }
        Else If status < sizeof(RTmr_val)
        {
                Display on screen “Partial RTmr received: (status) bytes”
                Exit Program
        }
        //Else it was successful
        RTmr_count++
        Write RTmr_val to binary file RTmr_File
        Reset timer
}

//Send count of entries received for check
Make all of the following writes blocking
Status = Send_UInt_To_Socket(RTmr_count)
If errno != 0
{
        Display on screen “Error in sending RTmr count: (errno)”
        Write_All_Output_To_File()
        Exit Program
}
Else If status != 4
{
        Display on screen “Write returned status of (status) on RTmr count”
        Write_All_Output_To_File()
        Exit Program
}

////////////////////////////////////////////////////////////////////////////////
//Receive SMon data and write to file
Make all of the following reads non-blocking
Start timer 5 – SMon count
Initialize status = -1
While status = -1
{
        Status = Read_UInt_From_Socket(Cli_SMon_count)
        If (status == -1) and (timer >= TIMEOUT_SMON_CNT)
        {
                Display on screen “Timeout occurred while waiting for SMon count”
                Exit Program
        }
```

```
}
If status = 0
{
    Display on screen “Socket was closed prematurely when trying to read SMon count”
    Exit Program
}
Else If status < sizeof(Cli_SMon_count)
{
    Display on screen “Partial SMon count received: (status) bytes”
    Exit Program
}

Start timer 6 – SMon
SMon_count = 0
While (SMon_count < Cli_SMon_count)
{
    Initialize status = -1
    While status = -1
    {
        Status = Read_USInt_From_Socket(SMon_val)
        If (status == -1) and (timer >= TIMEOUT_DATA_WORD)
        {
            Display on screen “Timeout occurred while waiting for SMon [SMon_count]”
            Exit Program
        }
    }
    If status = 0
    {
        Display on screen “Socket was closed prematurely when trying to read SMon [SMon_count]”
        Exit Program
    }
    Else If status < sizeof(SMon_val)
    {
        Display on screen “Partial SMon received: (status) bytes”
        Exit Program
    }
    //Else it was successful
    SMon_count++
    Write SMon_val to binary file SMon_File
    Reset timer
}

//Send count of entries received for check
Make all of the following writes blocking
Status = Send_UInt_To_Socket(SMon_count)
If errno != 0
{
    Display on screen “Error in sending SMon count: (errno)”
    Write_All_Output_To_File()
    Exit Program
}
Else If status != 4
{
    Display on screen “Write returned status of (status) on SMon count”
    Write_All_Output_To_File()
    Exit Program
}

Close files and socket
```

C.3. PTC Test Execution Software

```
Create a socket
Resolve server address
Connect to server
Enable standard I/O on socket
Make all of the following reads non-blocking

////////////////////////////////////////////////////////////////////////////////
//Receive Test Spec data
Start Timeout1(TIMEOUT_TS_CHAR_CNT)
Initialize status = -1
While (status = -1)
{
    Status = Read_UInt_From_Socket(Srv_TS_char_count)
    If ((status = -1) AND (Timeout1))
    {
        Display on screen "Timeout occurred while waiting for Test Spec character count"
        Exit Program
    }
}
If (status = 0)
{
    Display on screen "Socket was closed prematurely when trying to read Test Spec character count"
    Exit Program
}
Else If (status < sizeof(Srv_TS_char_count))
{
    Display on screen "Partial Test Spec character count received: (status) bytes"
    Exit Program
}
Else If (Srv_TS_char_count > MAX_TS_CHARS)
{
    Display on screen "Received Test Spec character count of (Srv_TS_char_count) exceeds maximum"
    Exit Program
}

Start Timeout2(TIMEOUT_DATA_WORD)
TS_char_count = 0
While (TS_char_count < Srv_TS_char_count)
{
    Initialize status = -1
    While (status = -1)
    {
        Status = Read_Char_From_Socket(spec_setup_lines[TS_char_count])
        If ((status = -1) AND (Timeout2))
        {
            Display on screen "Timeout occurred while waiting for Test Spec data character [TS_char_count]"
```

```
                Exit Program
            }
        }
        If (status = 0)
        {
            Display on screen "Socket was closed prematurely when trying to read Test Spec character"
            Exit Program
        }
        Else If (status < sizeof(spec_setup_lines[TS_char_count]))
        {
            Display on screen "Partial Test Spec data character [TS_char_count] received: (status) bytes"
            Exit Program
        }
        //Else it was successful
        TS_char_count++
        Reset Timeout2
    }
    //Send count of characters received for check
    Make all of the following writes blocking
    Status = Send_UInt_To_Socket(TS_char_count)
    If (errno != 0)
    {
        Display on screen "Error in sending Test Spec count: (errno)"
        Exit Program
    }
    Else If (status != 4)
    {
        Display on screen "Write returned status of (status) on Test Spec count"
        Exit Program
    }

    ////////////////////////////////////////////////////////////////////////////////
    //Receive fault vector data
    Make all of the following reads non-blocking
    Start Timeout3(TIMEOUT_FV_CNT)
    Initialize status = -1
    While (status = -1)
    {
        Status = Read_UInt_From_Socket(Srv_FV_count)
        If ((status = -1) AND (Timeout3))
        {
            Display on screen "Timeout occurred while waiting for fault vector count"
            Exit Program
        }
    }
    If (status = 0)
    {
        Display on screen "Socket was closed prematurely when trying to read fault vector count"
        Exit Program
    }
    Else If (status < sizeof(Srv_FV_count))
    {
        Display on screen "Partial fault vector count received: (status) bytes"
        Exit Program
    }
```

```
Else If (Srv_FV_count > MAX_FVS)
{
    Display on screen “Received fault vector count of (Srv_FV_count) exceeds maximum”
    Exit Program
}

Start Timeout4(TIMEOUT_DATA_WORD)
FV_count = 0
While (FV_count < Srv_FV_count)
{
    Initialize status = -1
    While (status = -1)
    {
        Status = Read_USInt_From_Socket(fault_vector[FV_count])
        If ((status = -1) AND (Timeout4))
        {
            Display on screen “Timeout occurred while waiting for fault vectors”
            Exit Program
        }
    }
    If (status = 0)
    {
        Display on screen “Socket was closed prematurely when trying to read fault vectors”
        Exit Program
    }
    Else If (status < sizeof(fault_vector[FV_count]))
    {
        Display on screen “Partial fault vector [FV_count] received: (status) bytes”
        Exit Program
    }
    //Else it was successful
    FV_count++
    Reset Timeout4
}

//Send count of fault vector received for check
Make all of the following writes blocking
Status = Send_UInt_To_Socket(FV_count)
If (errno != 0)
{
    Display on screen “Error in sending fault vector count: (errno)”
    Exit Program
}
Else If (status != 4)
{
    Display on screen “Write returned status of (status) on fault vector count”
    Exit Program
}


///////////////////////////////////////////////////////////////////////////////
// Execute Test
Initialize FPGA (load bit file)
Functional HW Reset
Initialize screen
```

```
While NOT (Keyboard_Input = ‘y’)
{
    Key_Pressed = Check_Keyboard()
    If (Key_Pressed)
    {
        Keyboard_Input = Get_Char()
        If (Keyboard_Input = ‘n’)
            Exit Program
    }
}
Key_Pressed = 0

Clear CCL Buffers
Send Round_Begin to HW
Start Timer for half second intervals

While NOT CCP_Round_Finished
{
    Get RTmr_Word, FTmr_Word, SMon_Word, FMon_Word from HW
    SMon_Avail = SMon_Word and SMON_BUFF_REC_AVAIL_ALL_MASK
    FMon_Avail = FMon_Word and CC_OBS_BUFF_REC_AVAIL_ALL_MASK

    // Select buffers to service
    If (SERVICE_RTMR_BUFFER = 0)
        RTmr_ Word = 0
    If (SERVICE_FTMR_BUFFER = 0)
        FTmr_Word = 0
    If (SERVICE_SMON_BUFFER = 0)
        SMon_Word = 0
    If (SERVICE_FMON_BUFFER = 0)
        FMon_Word = 0

    //Is there any data in the serviced buffers?
    If ((RTmr _Word(RTMR_BUFF_REC_AVAIL_MASK) != 0) OR
       (FTmr_Word(FTMR_BUFF_REC_AVAIL_MASK) != 0) OR (SMon_Word != 0) OR
       (FMon_Word != 0))
        Buffers_Empty = FALSE
    Else
        Buffers_Empty = TRUE

    If ((Buffers_Empty = FALSE) AND (Buffer_Overflow_Stop = FALSE))
    {
        //FMon
        If (FMon_Word(CC_OBS_BUFF_OVERFLOW_MASK) = 1)
        {
            FMon_Overflow = TRUE
            Write “FMon buffer Overflow” to Test_Log_File_Data
        }
        Else
            FMon_Overflow = FALSE

        All_Lanes_Data = TRUE
        For each I = 0 to Enabled_PEs – 1
        {
```

```
        BIU_Avail = CC_OBS_BUFF_REC_AVAIL_PE_MASK[I] and FMon_Word
        If (BIU_Avail <= 0)
            All_Lanes_Data = FALSE
    }

    If (All_Lanes_Data)
    {
        Increment Count_FV_Results
        For each I = 0 to Num_BIU
        {
            Choose PE FMon lane
            Write I to FMon_File_Data
            For J = 1 to (SWXF_Num_CC_Obs_DW + SWXF_Num_RTmr_IC_DW + 1)
            {
                Get FMon_Data from HW
                Write FMon_Data to FMon_File_Data
            }

            Partial_pop = FMon_Word and CC_OBS_BUFF_PARTIAL_POP_MASK

            If (Partial_pop != 0)
            {
                FMon_Partial_Pop = TRUE
                SW_End_Error_Flag = TRUE
                Write "FMon partial pop" to Test_Log_File_Data
                Write "SW End Round Error Detected" to SW_Error
            }
        }
    }

    //FTmr
    If (FTmr_Word(FTMR_BUFF_REC_AVAIL_MASK) = 1)
    {
        If (FTmr_Word(FTMR_BUFF_OVERFLOW_MASK) = 1)
        {
            FTmr_Overflow = TRUE
            Write "FTmr buffer Overflow" to Test_Log_File_Data
        }
        Else
            FTmr_Overflow = FALSE

        Choose FTmr in Data Port Select

        For I = 1 to (6 + SWXF_Num_RTmr_IC_DW)
        {
            Get FTmr_Data from HW
            Write FTmr_Data to FTmr_File_Data
        }

        Partial_pop = FTmr_Word and FTMR_BUFF_PARTIAL_POP_MASK

        If (Partial_pop != 0)
        {
            FTmr_Partial_Pop = TRUE
            SW_End_Error_Flag = TRUE
            Write “FTmr partial pop” to Test_Log_File_Data
            Write “SW End Round Error Detected” to SW_Error
        }
}

//RTmr
If (RTmr_Word(RTMR_BUFF_REC_AVAIL_MASK) = 1)
{
    If (RTmr_Word(RTMR_BUFF_OVERFLOW_MASK) = 1)
    {
        RTmr_Overflow = TRUE
        Write “RTmr buffer Overflow” to Test_Log_File_Data
    }
    Else
        FTmr_Overflow = FALSE

    Choose RTmr in Data Port Select

    For I = 1 to (1 + SWXF_Num_RTmr_IC_DW)
    {
        Get RTmr_Data from HW
        Write RTmr_Data to RTmr_File_Data
    }

    Partial_pop = RTmr_Word and RTMR_BUFF_PARTIAL_POP_MASK

    If (Partial_pop != 0)
    {
        RTmr_Partial_Pop = TRUE
        SW_End_Error_Flag = TRUE
        Write “RTmr partial pop” to Test_Log_File_Data
        Write “SW End Round Error Detected” to SW_Error
    }
}

//SMon
If (SMon_Word(SMON_BUFF_OVERFLOW_MASK) = 1)
{
    SMon_Overflow = TRUE
    Write “SMon buffer Overflow” to Test_Log_File_Data
}
Else
    SMon_Overflow = FALSE

For each I = 0 to Num_BIU
{
    BIU_Avail = SMON_BUFF_REC_AVAIL_PE_MASK[I] AND SMon_Word2

    If (BIU_Avail > 0)
    {
        Get Output_Enable = Data_Word and OUTPUT_ENA_ALL_MASK
        Data_Word = Output_Enable or SET_SMON_BUFF_SELECT_PE[I]
        Choose PE SMon lane
        Write I to SMon_File_Data
        For J = 1 to (PE_BIU_State_Msg_Length + SWXF_Num_RTmr_IC_DW + 2)
```

```
                {
                    Get SMon_Data from HW
                    Write SMon_Data to SMon_File_Data
                }

                Partial_pop = SMon_Word and SMON_BUFF_PARTIAL_POP_MASK

                If (Partial_pop != 0)
                {
                    SMon_Partial_Pop = TRUE
                    SW_End_Error_Flag = TRUE
                    Write "SMon partial pop" to Test_Log_File_Data
                    Write "SW End Round Error Detected" to SW_Error
                }
            }
        }

        //Set overflow stop condition
        If ((FMon_Overflow) OR (FTmr_Overflow) OR (RTmr_Overflow) OR (SMon_Overflow))
            Buffer_Overflow_Stop = 1
        Else
            Buffer_Overflow_Stop = 0
    }

    Else If (time_elapsed >= 0.5 seconds)
    {
        If (Next_Timed_Task = SCREEN_UPDATE)
        {
            update_screen()
            Next_Timed_Task = KEYBOARD_CHECK
        }
        Else //Next_Timed_Task = KEYBOARD_CHECK
        {
            Key_Pressed = Check_Keyboard()
            Next_Timed_Task = SCREEN_UPDATE
        }
        Reset time_elapsed
    }

    Else //CCP_State
    {
        Set RC_Status, FIC_Status, and SFT_Status from HW

        If (CCP_State = FINISH_UP_ROUND)
        {
            Write RC_Status to Test_Log_File_Data
            Stop_Condition = RC_Status and STOP_CONDITION_MASK
            Write Integer version of Stop_Condition to Test_Log_File_Data
            update_screen()
            Set CCP_Round_Finished to exit loop
        }

        Else
        {
            If (RC_Status(RND_STOP_MASK))
    CCP_State = FINISH_UP_ROUND

Else If ((Buffer_Overflow_Stop) OR (Key_Pressed) OR (SW_End_Error_Flag))
{
    Set Round Stop in HW
    CCP_State = WAIT_HW_ROUND_STOP
}

Else If (CCP_State = WAIT_HW_ROUND_STOP)
{
    If (RC_Status(RND_STOP_MASK))
        CCP_State = FINISH_UP_ROUND
}

Else If (CCP_State = WAIT_ROUND_ENABLE)
{
    If (RC_Status(RND_ENA_MASK))
        CCP_State = WAIT_SPIDER_READY
}

Else If (CCP_State = WAIT_SPIDER_READY)
{
    If (RC_Status(SPIDER_RDY_MASK))
        CCP_State = WAIT_FI_SETUP_END
}

Else If (CCP_State = WAIT_FI_SETUP_END)
{
    If (FIC_Status(FI_SETUP_END_MASK))
        CCP_State = WAIT_FN_SETUP_BEGIN
    Else
    {
        Get Spec_Token from Test_Spec Data
        If (Spec_Token = “No_FI”)
            Set No_FI_flag and FI_Setup_End in HW
        Else If (Spec_Token = “FI_Setup_Done”)
            Set FI_Setup_End in HW
        Else If (Spec_Token = “RCI_Test_Vector”)
            Write RCI_Test_Vec from Test_Spec data to HW
        Else If (Spec_Token = “RCI_All_Disable”)
            Write RCI_All_Dis from Test_Spec data to HW
        Else If ((Spec_Token = “FI_Fault”) OR (Spec_Token = “FI_Activation”))
        {
            Fault_Act_Field_Count = 0
            CCP_State = SEND_DATA_FI_SETUP
        }
        Else
        {
            Write “Wrong Test_Spec format - Expecting FI_Setup Line.” to Test_Log_File_Data
            Write “SW End Round Error Detected” to SW_Error
            Set Round Stop in HW
            CCP_State = WAIT_HW_ROUND_STOP
        }
    }
}
```

```
Else If (CCP_State = SEND_DATA_FI_SETUP)
{
    If (NOT (FIC_Status(FI_DATA_BUFF_FULL_MASK)))
    {
        If (NOT the End_of_Line(Test_Spec))
        {
            Get the next Data_Word from Test_Spec and send to HW
            Increment Fault_Act_Field_Count
        }
        Else //it is the end of the line
            CCP_State = SEND_SUMMARY_FI_SETUP
    }
}

Else If (CCP_State = SEND_SUMMARY_FI_SETUP)
{
    If (NOT (FIC_Status(FI_SUMMARY_BUFF_FULL_MASK)))
    {
        Convert Fault_Act_Field_Count to binary
        Data_Word = Shift binary value to FI_MSG_LENGTH_MASK position
        Write Data_Word to Summary Buffer of HW
        CCP_State = WAIT_FI_SETUP_END
    }
}

Else If (CCP_State = WAIT_FN_SETUP_BEGIN)
{
    If (SFT_Status(FUNC_SETUP_BEGIN_MASK))
        CCP_State = LOAD_FN_SETUP_APP_ASSIG
}

Else If (CCP_State = LOAD_FN_SETUP_APP_ASSIG)
{
    Get Spec_Token from Test_Spec Data
    If (Spec_Token = "App_Assignments")
    {
        Get App_Assig value from Test_Spec Data
        App_Assig_Read = TRUE
        Choose Application Assignment in Data Port Select of HW
        Write App_Assig to HW
        CCP_State = LOAD_FN_SETUP_RC_PER_CC
    }
    Else
    {
        Write "Wrong Test_Spec format - Expecting App_Assignments Line." To
            Test_Log_File_Data
        Write "SW End Round Error Detected" to SW_Error
        Set Round Stop in HW
        CCP_State = WAIT_HW_ROUND_STOP
    }
}

Else If (CCP_State = LOAD_FN_SETUP_RC_PER_CC)
{
    Get Spec_Token from Test_Spec Data
    If (Spec_Token = “RC_per_CC”)
    {
        Get RC_per_CC value from Test_Spec Data
        Choose RC_per_CC in Data Port Select of HW
        Write RC_per_CC to HW
        CCP_State = LOAD_FN_SETUP_CC_PER_RND
    }
    Else
    {
        Write “Wrong Test_Spec format - Expecting RC per CC Line.” To Test_Log_File_Data
        Write “SW End Round Error Detected” to SW_Error
        Set Round Stop in HW
        CCP_State = WAIT_HW_ROUND_STOP
    }
}

Else If (CCP_State = LOAD_FN_SETUP_CC_PER_RND)
{
    Get Spec_Token from Test_Spec Data
    If (Spec_Token = “CC_per_Round”)
    {
        Get CC_per_Rnd value from Test_Spec Data
        Choose CC_per_Rnd in Data Port Select of HW
        Write CC_per_Rnd to HW
        CCP_State = LOAD_FN_SETUP_ SCHEDULES
    }
    Else
    {
        Write “Wrong Test_Spec format - Expecting CC per Round Line.” To Test_Log_File_Data
        Write “SW End Round Error Detected” to SW_Error
        Set Round Stop in HW
        CCP_State = WAIT_HW_ROUND_STOP
    }
}

Else If (CCP_State = LOAD_FN_SETUP_SCHEDULES)
{
    // choose schedule in data port select and clear buffers
    Data_Word = SET_DATA_PORT_SELECT_SCHED OR CLR_SCHED_BUFF_MASK OR
                    CLR_SENSOR_BUFF_MASK OR CLR_COMMAND_BUFF_MASK
    Write Data_Word to HW
    Get Spec_Token from Test_Spec Data
    If (Spec_Token = “Schedule”)
    {
        While (Spec_Token = “Schedule”)
        {
            For each Num_BIU+1
            {
                Get Schedule_Field from Test_Spec Data
                Write Schedule_Field to HW
            }
            Get Spec_Token from Test_Spec Data
        }
        CCP_State = LOAD_FN_SETUP_SENSORS
    }
    Else
    {
        Write "Wrong Test_Spec format - Expecting Schedule Line." To Test_Log_File_Data
        Write "SW End Round Error Detected" to SW_Error
        Set Round Stop in HW
        CCP_State = WAIT_HW_ROUND_STOP
    }
}

Else If (CCP_State = LOAD_FN_SETUP_SENSORS)
{
    If (Spec_Token = "Sensors")
    {
        While not End_of_Line(Test_Spec)
        {
            Get Sensors_Field from Test_Spec Data
            Write Sensors_Field to HW
        }
        CCP_State = LOAD_FN_SETUP_COMMANDS
    }
    Else
    {
        Write "Wrong Test_Spec format - Expecting Sensors Line." To Test_Log_File_Data
        Write "SW End Round Error Detected" to SW_Error
        Set Round Stop in HW
        CCP_State = WAIT_HW_ROUND_STOP
    }
}

Else If (CCP_State = LOAD_FN_SETUP_COMMANDS)
{
    Get Spec_Token from Test_Spec Data
    If (Spec_Token = "Commands")
    {
        While not End_of_Line(Test_Spec)
        {
            Get Commands_Field from Test_Spec Data
            Write Commands_Field to HW
        }
        Preload_Count = 0
        Set Function_Setup_End in HW
        CCP_State = PRELOAD_FAULT_VECTORS
    }
    Else
    {
        Write "Wrong Test_Spec format - Expecting Commands Line." To Test_Log_File_Data
        Write "SW End Round Error Detected" to SW_Error
        Set Round Stop in HW
        CCP_State = WAIT_HW_ROUND_STOP
    }
}

Else If (CCP_State = PRELOAD_FAULT_VECTORS)
{
```

```
    If ((Preload_Count != PRELOAD_FAULT_TOTAL) AND
       (NOT Preload_End_of_File_Flag))
    {
        If (FV_Index < FV_count)
        {
            Increment FV_Index, SW_CCI, and Preload_Count
            Data_Word = FI_FIRE_TAG OR BROADCAST //header
            Send Data_Word to HW
            CCP_State = SEND_FIRE_ID
        }
        Else //end of file
            Preload_End_of_File_Flag = 1
    }
    Else //once the fault vectors have been preloaded
    {
        Preload_Flag = 0
        Set Function_Setup_End in HW to exit func setup phase (start func exec)
        CCP_State = WAIT_FN_ENABLE
    }
}

Else If (CCP_State = WAIT_FN_ENABLE)
{
    If (RC_Status(FUNC_ENA_MASK) = 1)
    {
        If (Preload_End_of_File_Flag = 0)
            CCP_State = SEND_FAULTLOAD
        Else
        {
            Set Control_Data_End in HW
            CCP_State = WAIT_HW_ROUND_STOP
        }
    }
}

Else If (CCP_State = SEND_FAULTLOAD)
{
    If (FIC_Status(FI_DATA_BUFF_FULL_MASK) = 0) // not full
    {
        If (FV_Index < FV_count)
        {
            Get Fault_Vector from Test_Spec Data
            Increment FV_Index and SW_CCI
            Data_Word = FI_FIRE_TAG OR Broadcast
            Send Data_Word to HW
            CCP_State = SEND_FIRE_ID
        }
        Else //end of file
        {
            Set Control_Data_End in HW
            CCP_State = WAIT_HW_ROUND_STOP
        }
    }
}
```

```
                Else If (CCP_State = SEND_FIRE_ID)
                {
                    If (FIC_Status(FI_DATA_BUFF_FULL_MASK) = 0) // not full
                    {
                        Write FIRE_ID to HW
                        CCP_State = SEND_DATA_FAULTLOAD
                    }
                }

                Else If (CCP_State = SEND_DATA_FAULTLOAD)
                {
                    If (FIC_Status(FI_DATA_BUFF_FULL_MASK) = 0) // not full
                    {
                        Write Fault_Vector to HW
                        CCP_State = SEND_SUMMARY_FAULTLOAD
                    }
                }

                Else If (CCP_State = SEND_SUMMARY_FAULTLOAD)
                {
                    If (FIC_Status(FI_SUMMARY_BUFF_FULL_MASK) = 0) // not full
                    {
                        Data_Word = FI_EXEC_FLAG_MASK OR Shift binary value of 3 to
                                        FI_MSG_LENGTH_MASK position
                        Write Data_Word to HW
                        If (Preload_Flag = 1)
                            CCP_State = PRELOAD_FAULT_VECTORS
                        Else
                            CCP_State = SEND _FAULTLOAD
                    }
                }
            }
        }
    }
```

//
//Send Test_Log_File_Data

```
Status = Send_UInt_To_Socket(TL_char_count)
If (errno != 0)
{
    Display on screen "Error in sending test log count: (errno)"
    Write_All_Output_To_File()
    Exit Program
}
Else If (status != 4)
{
    Display on screen "Write returned status of (status) on test log count"
    Write_All_Output_To_File()
    Exit Program
}

For each character in Test Log //if there are none, it will go on
    Send_Char_To_Socket(TL_char[])

//Check that the correct number of Test Log characters was received at server
```

```
Make all of the following reads non-blocking
Start Timeout5(TIMEOUT_SRV_TL _CNT)
Initialize status = -1
While (status = -1)
{
    Status = Read_UInt_From_Socket(Srv_TL_char_count)
    If ((status = -1) AND (Timeout5))
    {
        Display on screen "Timeout occurred while waiting for Test Log character count"
        Write_All_Output_To_File()
        Exit Program
    }
}
If (status = 0)
{
    Display on screen "Socket was closed prematurely when trying to read Test Log character count"
    Write_All_Output_To_File()
    Exit Program
}
Else If (status < sizeof(Srv_TL_char_count))
{
    Display on screen "Partial Test Log character count received: (status) bytes"
    Write_All_Output_To_File()
    Exit Program
}
Else If (Srv_TL_char_count != TL_char_count) //check that server got all
{
    Display on screen "Server received (Srv_TL_char_count) Test Spec characters when (TL_char_count) were
        sent"
    Write_All_Output_To_File()
    Exit Program
}
//Else it was successful

////////////////////////////////////////////////////////////////////////////////
//Send FMon File Data
Make all of the following writes blocking
Status = Send_UInt_To_Socket(FMon_count)
If (errno != 0)
{
    Display on screen "Error in sending FMon count: (errno)"
    Write_All_Output_To_File()
    Exit Program
}
Else If (status != 4)
{
    Display on screen "Write returned status of (status) on FMon count"
    Write_All_Output_To_File()
    Exit Program
}

For each entry in FMon //if there are none, it will go on
{
    Send_USInt_To_Socket(FMon_val[])
    If (errno != 0)
```

```
        {
            Display on screen “Error in sending FMon data: (errno)”
            Write_All_Output_To_File()
            Exit Program
        }
        Else If (status != 2)
        {
            Display on screen “Write returned status of (status) on FMon data”
            Write_All_Output_To_File()
            Exit Program
        }
    }

    //Check that the correct number of FMon was received at server
    Make all of the following reads non-blocking
    Start Timeout6 (TIMEOUT_SRV_FMON _CNT)
    Initialize status = -1
    While status = -1
    {
        Status = Read_UInt_From_Socket(Srv_FMon_count)
        If ((status = -1) AND (Timeout6))
        {
            Display on screen “Timeout occurred while waiting for FMon count”
            Write_All_Output_To_File()
            Exit Program
        }
    }
    If (status = 0)
    {
        Display on screen “Socket was closed prematurely when trying to read FMon count”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (status < sizeof(Srv_FMon_count))
    {
        Display on screen “Partial FMon count received: (status) bytes”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (Srv_FMon_count != FMon_count) //check that server got all
    {
        Display on screen “Server received (Srv_ FMon _count) FMon entries when (FMon _count) were sent”
        Write_All_Output_To_File()
        Exit Program
    }
    //Else it was successful

    ////////////////////////////////////////////////////////////////////////////////
    //Send FTmr File Data
    Make all of the following writes blocking
    Status = Send_UInt_To_Socket(FTmr_count)
    If (errno != 0)
    {
        Display on screen “Error in sending FTmr count: (errno)”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (status != 4)
    {
        Display on screen “Write returned status of (status) on FTmr count”
        Write_All_Output_To_File()
        Exit Program
    }

    For each entry in FTmr //if there are none, it will go on
    {
        Send_USInt_To_Socket(FTmr_val[])
        If (errno != 0)
        {
            Display on screen “Error in sending FTmr data: (errno)”
            Write_All_Output_To_File()
            Exit Program
        }
        Else If (status != 2)
        {
            Display on screen “Write returned status of (status) on FTmr data”
            Write_All_Output_To_File()
            Exit Program
        }
    }

    //Check that the correct number of FTmr was received at server
    Make all of the following reads non-blocking
    Start Timeout7 (TIMEOUT_SRV_FTMR _CNT)
    Initialize status = -1
    While status = -1
    {
        Status = Read_UInt_From_Socket(Srv_FTmr_count)
        If ((status = -1) AND (Timeout7))
        {
            Display on screen “Timeout occurred while waiting for FTmr count”
            Write_All_Output_To_File()
            Exit Program
        }
    }
    If (status = 0)
    {
        Display on screen “Socket was closed prematurely when trying to read FTmr count”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (status < sizeof(Srv_FTmr_count))
    {
        Display on screen “Partial FTmr count received: (status) bytes”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (Srv_FTmr_count != FTmr_count) //check that server got all
    {
        Display on screen “Server received (Srv_FTmr_count) FTmr entries when (FTmr_count) were sent”
        Write_All_Output_To_File()
        Exit Program
    }
    //Else it was successful

    ////////////////////////////////////////////////////////////////////////////////
    //Send RTmr File Data
    Make all of the following writes blocking
    Status = Send_UInt_To_Socket(RTmr_count)
    If (errno != 0)
    {
        Display on screen "Error in sending RTmr count: (errno)"
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (status != 4)
    {
        Display on screen "Write returned status of (status) on RTmr count"
        Write_All_Output_To_File()
        Exit Program
    }

    For each entry in RTmr //if there are none, it will go on
    {
        Send_USInt_To_Socket(RTmr_val[])
        If (errno != 0)
        {
            Display on screen "Error in sending RTmr data: (errno)"
            Write_All_Output_To_File()
            Exit Program
        }
        Else If (status != 2)
        {
            Display on screen "Write returned status of (status) on RTmr data"
            Write_All_Output_To_File()
            Exit Program
        }
    }

    //Check that the correct number of RTmr was received at server
    Make all of the following reads non-blocking
    Start Timeout8 (TIMEOUT_SRV_RTMR _CNT)
    Initialize status = -1
    While status = -1
    {
        Status = Read_UInt_From_Socket(Srv_RTmr_count)
        If ((status = -1) AND (Timeout8))
        {
            Display on screen "Timeout occurred while waiting for RTmr count"
            Write_All_Output_To_File()
            Exit Program
        }
    }
    If (status = 0)
    {
        Display on screen “Socket was closed prematurely when trying to read RTmr count”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (status < sizeof(Srv_RTmr_count))
    {
        Display on screen “Partial RTmr count received: (status) bytes”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (Srv_RTmr_count != RTmr_count) //check that server got all
    {
        Display on screen “Server received (Srv_RTmr_count) RTmr entries when (RTmr_count) were sent”
        Write_All_Output_To_File()
        Exit Program
    }
    //Else it was successful

    /////////////////////////////////////////////////////////////////////////////////
    //Send SMon File Data
    Make all of the following writes blocking
    Status = Send_UInt_To_Socket(SMon_count)
    If (errno != 0)
    {
        Display on screen “Error in sending SMon count: (errno)”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (status != 4)
    {
        Display on screen “Write returned status of (status) on SMon count”
        Write_All_Output_To_File()
        Exit Program
    }

    For each entry in SMon //if there are none, it will go on
    {
        Send_USInt_To_Socket(SMon_val[])
        If (errno != 0)
        {
            Display on screen “Error in sending SMon data: (errno)”
            Write_All_Output_To_File()
            Exit Program
        }
        Else If (status != 2)
        {
            Display on screen “Write returned status of (status) on SMon data”
            Write_All_Output_To_File()
            Exit Program
        }
    }

    //Check that the correct number of SMon was received at server
    Make all of the following reads non-blocking
    Start Timeout9 (TIMEOUT_SRV_SMON _CNT)
```

```
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Srv_SMon_count)
    If ((status = -1) AND (Timeout9))
    {
        Display on screen "Timeout occurred while waiting for SMon count"
        Write_All_Output_To_File()
        Exit Program
    }
}
If (status = 0)
{
    Display on screen "Socket was closed prematurely when trying to read SMon count"
    Write_All_Output_To_File()
    Exit Program
}
Else If (status < sizeof(Srv_SMon_count))
{
    Display on screen "Partial SMon count received: (status) bytes"
    Write_All_Output_To_File()
    Exit Program
}
Else If (Srv_SMon_count != SMon_count) //check that server got all
{
    Display on screen "Server received (Srv_SMon_count) SMon entries when (SMon_count) were sent"
    Write_All_Output_To_File()
    Exit Program
}
//Else it was successful

Close socket

//////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Write_All_Output_To_File()
//Test_Log_File
Open Test Log File
Write_string(Test_Log_File_Data, TL_char)

//FMon_File
Open FMon File
Index = 0
For index < FMon_count
{
    Write_binary(FMon_File, FMon_val[index])
    Index++
}

//FTmr_File
Open FTmr File
Index = 0
For index < FTmr_count
{
    Write_binary(FTmr_File, FTmr_val[index])
    Index++
}

//RTmr_File
Open RTmr File
Index = 0
For index < RTmr_count
{
    Write_binary(RTmr_File, RTmr_val[index])
    Index++
}

//SMon_File
Open SMon File
Index = 0
For index < SMon_count
{
    Write_binary(SMon_File, SMon_val[index])
    Index++
}
Close files (Test_Log_File, FMon_File, FTmr_File, RTmr_File, SMon_File)
```

C.4. STC Test Execution Software

```
Create a socket
Resolve server address
Connect to server
Enable standard I/O on socket

Initialize FPGA (load bit file)
Functional HW Reset
Initialize screen

While NOT (Keyboard_Input = 'y')
{
    Key_Pressed = Check_Keyboard()
    If (Key_Pressed)
    {
        Keyboard_Input = Get_Char()
        If (Keyboard_Input = 'n')
            Exit Program
    }
}
Key_Pressed = 0

Clear CCL Buffers
Send Round_Begin to HW
Start Timer for half second intervals

While NOT CCP_Round_Finished
{
    Get RTmr_Word and SMon_Word from HW
    SMon_Avail = SMon_Word and SMON_BUFF_REC_AVAIL_ALL_MASK
```

```
// Select buffers to service
If (SERVICE_RTMR_BUFFER = 0)
    RTmr_ Word = 0
If (SERVICE_SMON_BUFFER = 0)
    SMon_Word = 0

//Is there any data in the serviced buffers?
If ((RTmr_Word(RTMR_BUFF_REC_AVAIL_MASK) != 0) OR (SMon_Word != 0))
    Buffers_Empty = FALSE
Else
    Buffers_Empty = TRUE

If (Buffers_Empty = FALSE)
{
    //RTmr
    If (RTmr_Word(RTMR_BUFF_REC_AVAIL_MASK) = 1)
    {
        If (RTmr_Word(RTMR_BUFF_OVERFLOW_MASK) = 1)
        {
            RTmr_Overflow = TRUE
            Write "RTmr buffer Overflow" to Test_Log_File_Data
        }
        Else
            FTmr_Overflow = FALSE

        Choose RTmr in Data Port Select

        For I = 1 to (1 + SWXF_Num_RTmr_IC_DW)
        {
            Get RTmr_Data from HW
            Write RTmr_Data to RTmr_File_Data
        }

        Partial_pop = RTmr_Word and RTMR_BUFF_PARTIAL_POP_MASK

        If (Partial_pop != 0)
        {
            RTmr_Partial_Pop = TRUE
            SW_End_Error_Flag = TRUE
            Write "RTmr partial pop" to Test_Log_File_Data
            Write "SW End Round Error Detected" to SW_Error
        }
    }

    //SMon
    If (SMon_Word(SMON_BUFF_OVERFLOW_MASK) = 1)
    {
        SMon_Overflow = TRUE
        Write "SMon buffer Overflow" to Test_Log_File_Data
    }
    Else
        SMon_Overflow = FALSE

    For each I = 0 to Num_BIU
    {
        BIU_Avail = SMON_BUFF_REC_AVAIL_PE_MASK[I] AND SMon_Word2

        If (BIU_Avail > 0)
        {
            Get Output_Enable = Data_Word and OUTPUT_ENA_ALL_MASK
            Data_Word = Output_Enable or SET_SMON_BUFF_SELECT_PE[I]
            Choose PE SMon lane
            Write I to SMon_File_Data
            For J = 1 to (PE_BIU_State_Msg_Length + SWXF_Num_RTmr_IC_DW + 2)
            {
                Get SMon_Data from HW
                Write SMon_Data to SMon_File_Data
            }

            Partial_pop = SMon_Word and SMON_BUFF_PARTIAL_POP_MASK

            If (Partial_pop != 0)
            {
                SMon_Partial_Pop = TRUE
                SW_End_Error_Flag = TRUE
                Write "SMon partial pop" to Test_Log_File_Data
                Write "SW End Round Error Detected" to SW_Error
            }
        }
    }

    //Set overflow stop condition
    If ((RTmr_Overflow) OR (SMon_Overflow))
        Buffer_Overflow_Stop = 1
    Else
        Buffer_Overflow_Stop = 0
}

Else If (time_elapsed >= 0.5 seconds)
{
    If (Next_Timed_Task = SCREEN_UPDATE)
    {
        update_screen()
        Next_Timed_Task = KEYBOARD_CHECK
    }
    Else //Next_Timed_Task = KEYBOARD_CHECK
    {
        Key_Pressed = Check_Keyboard()
        Next_Timed_Task = SCREEN_UPDATE
    }
    Reset time_elapsed
}

Else //CCP_State
{
    Set RC_Status and FIC_Status from HW

    If (CCP_State = FINISH_UP_ROUND)
    {
        Write RC_Status to Test_Log_File_Data
```

```
                Stop_Condition = RC_Status and STOP_CONDITION_MASK
                Write Integer version of Stop_Condition to Test_Log_File_Data
                update_screen()
                Set CCP_Round_Finished to exit loop
            }

            Else
            {
                If (RC_Status(RND_STOP_MASK))
                    CCP_State = FINISH_UP_ROUND

                Else If ((Buffer_Overflow_Stop) OR (Key_Pressed) OR (SW_End_Error_Flag))
                {
                    Set Round Stop in HW
                    CCP_State = WAIT_HW_ROUND_STOP
                }
                Else If (CCP_State = WAIT_HW_ROUND_STOP)
                {
                    If (RC_Status(RND_STOP_MASK))
                        CCP_State = FINISH_UP_ROUND
                }
                Else If (CCP_State = WAIT_ROUND_ENABLE)
                {
                    If (RC_Status(RND_ENA_MASK))
                    {
                        Choose Round Index in Data Port Select in HW
                        Get Round_Index from HW
                        Choose Enabled Nodes in Data Port Select in HW
                        Get Enabled_Nodes from HW
                        Set Num_RMU based on Enabled_Nodes
                        Enable SMon Output lanes based on Enabled_Nodes
                        CCP_State = WAIT_SPIDER_READY
                    }
                }
                Else If (CCP_State = WAIT_SPIDER_READY)
                {
                    If (RC_Status(SPIDER_RDY_MASK))
                        CCP_State = WAIT_FI_SETUP_END
                }
                Else If (CCP_State = WAIT_FI_SETUP_END)
                {
                    If (FIC_Status(FI_SETUP_END_MASK))
                        CCP_State = WAIT_FN_ENABLE
                    Else
                        Set FI_Setup_End in HW
                }
                Else If (CCP_State = WAIT_FN_ENABLE)
                {
                    If (RC_Status(FUNC_ENA_MASK) = 1)
                        CCP_State = WAIT_HW_ROUND_STOP
                }
            }
        }
}
```

```
///////////////////////////////////////////////////////////////////////////////
//Send Test_Log_File_Data
Status = Send_UInt_To_Socket(TL_char_count)
If (errno != 0)
{
    Display on screen "Error in sending test log count: (errno)"
    Write_All_Output_To_File()
    Exit Program
}
Else If (status != 4)
{
    Display on screen "Write returned status of (status) on test log count"
    Write_All_Output_To_File()
    Exit Program
}

For each character in Test Log //if there are none, it will go on
    Send_Char_To_Socket(TL_char[])

//Check that the correct number of Test Log characters was received at server
Make all of the following reads non-blocking
Start Timetou1 (TIMEOUT_SRV_TL _CNT)
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Srv_TL_char_count)
    If ((status = -1) and (Timeout1))
    {
        Display on screen "Timeout occurred while waiting for Test Log character count"
        Write_All_Output_To_File()
        Exit Program
    }
}
If (status = 0)
{
    Display on screen "Socket was closed prematurely when trying to read Test Log character count"
    Write_All_Output_To_File()
    Exit Program
}
Else If (status < sizeof(Srv_TL_char_count))
{
    Display on screen "Partial Test Log character count received: (status) bytes"
    Write_All_Output_To_File()
    Exit Program
}
Else If (Srv_TL_char_count != TL_char_count) //check that server got all
{
    Display on screen "Server received (Srv_TL_char_count) Test Spec characters when (TL_char_count) were
        sent"
    Write_All_Output_To_File()
    Exit Program
}
//Else it was successful

///////////////////////////////////////////////////////////////////////////////
```

```
**//Send RTmr File Data**
Make all of the following writes blocking
Status = Send_UInt_To_Socket(RTmr_count)
If (errno != 0)
{
    Display on screen “Error in sending RTmr count: (errno)”
    Write_All_Output_To_File()
    Exit Program
}
Else If (status != 4)
{
    Display on screen “Write returned status of (status) on RTmr count”
    Write_All_Output_To_File()
    Exit Program
}

For each entry in RTmr //if there are none, it will go on
{
    Send_USInt_To_Socket(RTmr_val[])
    If (errno != 0)
    {
        Display on screen “Error in sending RTmr data: (errno)”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (status != 2)
    {
        Display on screen “Write returned status of (status) on RTmr data”
        Write_All_Output_To_File()
        Exit Program
    }
}

//Check that the correct number of RTmr was received at server
Make all of the following reads non-blocking
Start Timeout2 (TIMEOUT_SRV_RTMR _CNT)
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Srv_RTmr_count)
    If ((status = -1) and (Timeout2))
    {
        Display on screen “Timeout occurred while waiting for RTmr count”
        Write_All_Output_To_File()
        Exit Program
    }
}
If (status = 0)
{
    Display on screen “Socket was closed prematurely when trying to read RTmr count”
    Write_All_Output_To_File()
    Exit Program
}
Else If (status < sizeof(Srv_RTmr_count))
{
    Display on screen “Partial RTmr count received: (status) bytes”
    Write_All_Output_To_File()
    Exit Program
}
Else If (Srv_RTmr_count != RTmr_count) //check that server got all
{
    Display on screen “Server received (Srv_RTmr_count) RTmr entries when (RTmr_count) were sent”
    Write_All_Output_To_File()
    Exit Program
}
//Else it was successful
```

//
//Send SMon File Data

```
Make all of the following writes blocking
Status = Send_UInt_To_Socket(SMon_count)
If (errno != 0)
{
    Display on screen “Error in sending SMon count: (errno)”
    Write_All_Output_To_File()
    Exit Program
}
Else If (status != 4)
{
    Display on screen “Write returned status of (status) on SMon count”
    Write_All_Output_To_File()
    Exit Program
}

For each entry in SMon //if there are none, it will go on
{
    Send_USInt_To_Socket(SMon_val[])
    If (errno != 0)
    {
        Display on screen “Error in sending SMon data: (errno)”
        Write_All_Output_To_File()
        Exit Program
    }
    Else If (status != 2)
    {
        Display on screen “Write returned status of (status) on SMon data”
        Write_All_Output_To_File()
        Exit Program
    }
}

//Check that the correct number of SMon was received at server
Make all of the following reads non-blocking
Start Timeout3 (TIMEOUT_SRV_SMON _CNT)
Initialize status = -1
While status = -1
{
    Status = Read_UInt_From_Socket(Srv_SMon_count)
    If ((status = -1) and (Timeout3))
    {
        Display on screen “Timeout occurred while waiting for SMon count”
        Write_All_Output_To_File()
        Exit Program
    }
}
If (status = 0)
{
    Display on screen “Socket was closed prematurely when trying to read SMon count”
    Write_All_Output_To_File()
    Exit Program
}
Else If (status < sizeof(Srv_SMon_count))
{
    Display on screen “Partial SMon count received: (status) bytes”
    Write_All_Output_To_File()
    Exit Program
}
Else If (Srv_SMon_count != SMon_count) //check that server got all
{
    Display on screen “Server received (Srv_SMon_count) SMon entries when (SMon_count) were sent”
    Write_All_Output_To_File()
    Exit Program
}
//Else it was successful

Close socket

/////////////////////////////////////////////////////////////////////////////////////////////////////////
//Write_All_Output_To_File()
//Test_Log_File
Open Test Log File
Write_string(Test_Log_File, TL_char)

//RTmr_File
Open RTmr File
Index = 0
For index < RTmr_count
{
    Write_binary(RTmr_File, RTmr_val[index])
    Index++
}

//SMon_File
Open SMon File
Index = 0
For index < SMon_count
{
    Write_binary(SMon_File, SMon_val[index])
    Index++
}
Close files (Test_Log_File, RTmr_File, SMon_File)
```

References

[ARINC651] ARINC 651-1: *Design Guidance for Integrated Modular Avionics*. Aeronautical Radio, Inc., November 7, 1997.

[Arlat03] Arlat, Jean; et al: *Comparison of Physical and Software-Implemented Fault Injection Techniques*. IEEE Transaction on Computers, Vol. 52, No. 9, September 2003, pp. 1115-1133.

[Arlat89] Arlat, Jean; et al: *Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems*. 19th International Symposium on Fault Tolerant Computing Symposium (FTCS 19), 1989, pp. 348-355.

[Armstrong93] Armstrong, James R.; and Gray, F. Gail: *Structured Logic Design with VHDL*. Prentice Hall PTR, 1993.

[Avizienis04] Avizienis, Algirdas; et al: *Basic Concepts and Taxonomy of Dependable and Secure Computing*. IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January-March 2004, pp. 11-33.

[Azadmanesh00] Azadmanesh, M.H.; and Kieckhafer, R.M.: *Exploiting omissive faults in synchronous approximate agreement*. IEEE Transaction on Computers, Vol. 49, No. 10, October 2000, pp. 1031–1042.

[Butler02] Butler, Ricky W.; et al: *NASA Langley's Research and Technology Transfer Program in Formal Methods*. NASA Langley Research Center, May 2002. Available on the World Wide Web at http://shemesh.larc.nasa.gov/fm/NASA-over.pdf.

[Butler08] Butler, Ricky W.; *A Primer on Architectural Level Fault Tolerance*. NASA/TM-2008-215108, 2008.

[Chávez10] Chávez-Fuentes, J. R.; González, O. R.; and Gray, W. S.: *Performance Analysis of Fault Tolerant Control Systems with I.I.D. Upsets.* Proc. 2010 American Control Conference, Baltimore, Maryland, 2010, to appear

[Clough96] Clough, B. T.: *Effects of electromagnetic interference on digital control systems*. Wright-Patterson AFB, Dayton, OH, Internal Report WL-TR-96-3122, 1996.

[DeMicheli94] De Micheli, Giovanni: *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[Driscoll03] Driscoll, Kevin; Hall, Brendan; Sivencrona, Hakan; and Zumsteg, Phil: *Byzantine Fault Tolerance, from Theory to Reality*. 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP03), Edinburgh, Scotland, UK, October 2003, pp. 235-248.

[FAA93] FAA: *Certification of aircraft electrical/electronic systems for operation in the high intensity radiated fields (HIRF) environment*. Federal Aviation Administration, Proposed FAA Circular ARD 50040 Draft 16, August 1993.

[Fuller95] Fuller, Gerald L.: *Understanding HIRF – High Intensity Radiated Fields*. Aviation Communications, Inc., Leesburg, VA, 1995, p. 7-2

[González01] González, O. R.; Gray, W. S.; and Tejada, A.: *Analytical Tools for the Design and Verification of Safety Critical Control Systems*. 2001 SAE Transactions - Journal of Aerospace, vol. 110, Section 1, 2001, pp. 481-490

[Gray00] Gray, W. S.; González, O. R.; and Dogan, M.: *Stability Analysis of Digital Linear Flight Controllers Subject to Electromagnetic Disturbances*. IEEE Transactions on Aerospace and Electronic Systems, vol. 36, no. 4, October 2000, pp. 1204-1218.

[Gray08] Gray, W. Steven; Wang, Rui; and González, Oscar R.: *A Performance Model for a Distributed Flight Control System Subject to Random Upsets*. Proc. 2008 IEEE Conference on Control Applications, San Antonio, Texas, 2008, pp. 918-923.

[Gray10] Gray, W. S.; Wang, R.; González, O. R.; and Chávez-Fuentes, J. R.: *Tracking Performance Analysis of a Distributed Recoverable Boeing 747 Flight Control System Subject to Digital Upsets*. Proc. 2010 American Control Conference, Baltimore, Maryland, 2010, to appear

[Hess97] Hess, Richard: *Computing Platform Architectures for Robust Operation in the Presence of Lightning and Other Electromagnetic Threats*. Presented at the 16th Digital Avionics Systems Conference (DASC), Irvine, California, October 26-30, 1997

[Hsueh97] Hsueh, Mei-Chen; Tsai, T.K.; and Iyer, R. K.: *Fault Injection Techniques and Tools*. Computer, Vol. 30, Issue 4, April 1997, pp. 75-82.

[IVHM08] *Integrated Vehicle Health Management, Technical Plan, Version 2.02*. NASA, Aeronautic Mission Research Directorate, Aviation Safety Program, December 8, 2008.

[Karlsson95] Karlsson, J.; et al: *Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture*. Proc. 5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5), IEEE Computer Society Press, Urbana-Champaign, IL, USA, September 1995, pp. 267-287.

[Koopman04] Koopman, Philip; and Chakavarty, Tridib: *Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks*. The International Conference on Dependable Systems and Networks (DSN), 2004, pp. 145 - 154.

[Lala91] Lala, Jaynarayan H.; Harper, Richard E.; and Alger, Linda S.: *A Design Approach for Ultrareliable Real-Time Systems*. IEEE Computer, Vol. 24, No. 5, May 1991, pp. 12-22.

[Layton98] Layton, P. J.; et al: *Single Event Latchup Protection of Integrated Circuits*. Fourth European Conference on radiation and Its Effects on Components and Systems, (RADECS 97), Cannes, France, 1997, pp. 327 - 331.

[Malekpour09] Malekpour, Mahyar R.: *A Self-Stabilizing Byzantine-Fault-Tolerant Clock Synchronization Protocol*. NASA/TM-2009-215758, June 2009, pp. 43.

[Miner02] Miner, Paul S.; Malekpour, Mahyar; and Torres, Wilfredo: *A Conceptual Design for a Reliable Optical Bus (ROBUS)*. Presented at the 21st Digital Avionics Systems Conference (DASC), Irvine, California, October 27-31, 2002.

[NASA06] *2006 NASA Strategic Plan*. NP-2006-02-423-HQ, 2006.

[Paulitsch05] Paulitsch, Michael; et al: *Coverage and the Use of Cyclic Redundancy Codes in Ultra-Dependable Systems*. International Conference on Dependable Systems and Networks (DSN), June 2005.

[PC/104] *PC/104-Plus Specification*. Version 1.1, PC/104 Consortium, June 1997.

[Powell92] Powell, David: *Failure Mode Assumptions and Assumption Coverage.* 22nd International Symposium on Fault Tolerant Computing Systems, 1992, pp. 396 - 395.

[R2PP] On the World Wide Web at http://opensource.arc.nasa.gov/project/robus-2/

[Ramabadran88] Ramabadran, Tenkasi V.; and Gaitonde, Sunil S.: *A Tutorial on CRC Computations*. IEEE Micro, Vol. 8, Issue 4, 1998, pp. 62 - 75.

[Shin08] Shin, Jaiwon. Associate Administrator, Aeronautic Mission Research Directorate. *Opening Remarks, Aviation Safety Program*. 2008 Annual Technical Meeting, Denver, CO, October 21, 2008.

[Stallings94] Stallings, William: *Data and Computer Communications*. 4th Edition, Prentice-Hall, Inc., 1994, p. 166.

[Taber93] Taber, A.; and Normand, E.: *Single Event Upset in Avionics*. IEEE Transactions on Nuclear Science, Vol. 40, No. 2, April 1993, pp. 120-126.

[Torres05A] Torres-Pomales, Wilfredo; Malekpour, Mahyar; and Miner, Paul S.: *ROBUS-2: A Fault-Tolerant Broadcast Communication System*. NASA TM-2005-213540, 2005.

[Torres05B] Torres-Pomales, Wilfredo; Malekpour, Mahyar; and Miner, Paul S.: *Design of the Protocol Processor for the ROBUS-2 Communication System*. NASA TM-2005-213934, 2005.

[Torres08A] Torres-Pomales, Wilfredo; Malekpour, Mahyar R.; Miner, Paul S.; and Koppen, Sandra V.: *Plan for the Characterization of HIRF Effects on a Fault-Tolerant Computer Communication System*. NASA/TM-2008-215306, 2008.

[Torres08B] Torres-Pomales, Wilfredo; Malekpour, Mahyar R.; Miner, Paul S.; and Koppen, Sandra V.: *Design of Test Articles and Monitoring System for the Characterization of HIRF Effects on a Fault-Tolerant Computer Communication System*. NASA/TM-2008-215322, 2008.

[Weber06] Weber, Paul J.: *Dynamic Reduction Algorithms for Fault Tolerant Convergent Voting with Hybrid Faults*. Doctoral Dissertation, Michigan Technological University, May 2006.

[XAPP077] Xilinx Application Note XAPP077: *Metastability Considerations*. Version 1.0, Xilinx Corporation, January 1997.

[XAPP230] Xilinx Application Note XAPP230: *The LVDS I/O Standard*. Version 1.1, Xilinx Corporation, November 1999.

[Yates10] Yates, Amy M.; Torres-Pomales, Wilfredo; Malekpour, Mahyar R.; González, Oscar R.; and Gray, W. Steven: *Design of a High-Intensity Radiated Field Fault-Injection Experiment for a Fault-Tolerant Distributed Computation and Communication System*. Proc. 2010 Digital Avionics Systems Conference (DASC), Salt Lake City, Utah, 2010, to appear.

[Zhang08] Zhang, H.; Gray, W. S.; and González, O. R.: *Performance Analysis of Digital Flight Control Systems with Rollback Error Recovery Subject to Simulated Neutron-Induced Upsets*. IEEE Transactions on Control Systems Technology, vol. 16, no. 1, January 2008, pp. 46-59.

[Zhang09] Zhang, H.; Gray, W. S.; González, O. R.; and Lakdawala, A. V.: *Output Tracking Performance of a Recoverable Digital Flight Control System in Neutron Environments*. IEEE Transactions on Aerospace and Electronic Systems, vol. 45, no. 1, January 2009, pp. 321-335.

Acronyms and Abbreviations

ASAP	As Soon As Possible
ASCII	American Standard Code for Information Interchange
BER	Bit Error Rate
BIU	Bus Interface Unit
CC	Control Cycle
CCI	Control Cycle Index
CCL	Controller Coordination Link
CCP	Controller Coordination Protocol
CDM	Clique Detection Mode
CDU	Correct, Detected, Undetected
CFIMS	Configurable Fault Injection and Monitoring System
CIM	Clique Initialization Mode
CJM	Clique Join Mode
COTS	Commercial Off-The-Shelf
CPLD	Complex Programmable Logic Device
CPM	Clique Preservation Mode
CRC	Cyclic Redundancy Code
CTS	Clear To Send
DCP	Detectability, Consistency and Persistence
DII	Data Introduction Interval
DL	Data Link
ECR	Error Containment Region
EMI	Electromagnetic Interference
ENM	Embedded Node Monitor
FCR	Fault Containment Region
FCS	Frame Check Sequence
FI	Fault Injection; Fault Injectors
FIC	Fault Injection Controller
FIFO	First-In, First-Out
FMon	Function Monitor
FPGA	Field-Programmable Gate Array
FTmr	Function Timer
GOT	Good, Omissive, Transmissive
HFCS	Header Frame Check Sequence
HIRF	High Intensity Radiated Field
HMon	Health Monitor
IC	Interval Count
IEV	Input Eligible Voters
IIEV	Initial Input Eligible Voters
IMA	Integrate Modular Avionics
IO	Input-Output
IORU	Input-Output Receive Unit
IOSU	Input-Output Send Unit
IT	Interval Time
IU	Input Unit
IVHM	Integrated Vehicle Health Management
LaRC	Langley Research Center
LSM	Lane State Monitor
LVDS	Low-Voltage Differential Signaling
M	Number of BIUs (equal in value to Num_RMU)
MBS	Manchester Bit Stream
MC	Manchester Code; Master Controller

MCU	Mode Control Unit
N	Number of BIUs (equal in value to Num_BIU and Num_PE)
NASA	National Aeronautics and Space Administration
NCM	Node Condition Monitor
NRZ	Non-Return-To-Zero
Num_BIU	Number of BIUs
Num_PE	Number of PEs
Num_RMU	Number of RMUs
OCL	Operation Coordination Level
ODU	Old Dominion University
OK	Opposite Kind
OTH	Omissive-Transmissive Hybrid
PE	Processing Element
PESU	PE Setup Unit
PFCS	Packet Frame Check Sequence
PFD	Packet Frame Decoder
PFE	Packet Frame Encoder
PISO	Parallel-In Serial-Out
PMCU	Packet-Mode Communication Unit
PTC	Primary Test Controller
PTL	Primary Test Link
RC	Round Control; ROBUS Cycle
RCI	Round Cycle Index
RCtlr	Round Controller
RL	ROBUS Link
RM	ROBUS Message
RMU	Redundancy Management Unit
ROBUS	Reliable Optical Bus (obsolete as an acronym; now ROBUS is the name given to the communication system concept); Robust Bus
RPP	ROBUS Protocol Processor
RRU	ROBUS Receive Unit
RSPP	Reconfigurable SPIDER Prototyping Platform
RSU	ROBUS Send Unit
RT	Round Time
RTmr	Round Timer
RTS	Request To Send
Rx	Receiver
SCP	Self-Checking Pair
SD	Signal Detect
SFT	SPIDER Function Tester
SK	Same Kind
SMon	State Monitor
SMR	State Message Receiver
SMU	State Monitoring Unit
SPIDER	Scalable Processor-Independent Design for Extended Reliability
SRAM	Static Random Access Memory
STC	Secondary Test Controller
STL	Secondary Test Link
STM	Self-Test Mode
STMon	Stop Trigger Monitor
SUT	System Under Test
SW	Software
SWXF	Software Interface
Sync	Synchronization
TCL	Test Control Link

TCM	Test Control Message
TDMA	Time-Division Multiple Access
Tx	Transmitter
VHDL	VHSIC (Very-High-Speed Integrated Circuit) Hardware Description Language
WFD	Word Frame Decoder
WFE	Word Frame Encoder
WMCU	Word-Mode Communication Unit

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 01- 08 - 2010	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Fault Injection and Monitoring Capability for a Fault-Tolerant Distributed Computation System	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Torres-Pomales, Wilfredo; Yates, Amy M.; Malekpour, Mahyar R.	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 645846.02.07.07.15.02
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER** L-19899
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/TM-2010-216834

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Configurable Fault-Injection and Monitoring System (CFIMS) is intended for the experimental characterization of effects caused by a variety of adverse conditions on a distributed computation system running flight control applications. A product of research collaboration between NASA Langley Research Center and Old Dominion University, the CFIMS is the main research tool for generating actual fault response data with which to develop and validate analytical performance models and design methodologies for the mitigation of fault effects in distributed flight control systems. Rather than a fixed design solution, the CFIMS is a flexible system that enables the systematic exploration of the problem space and can be adapted to meet the evolving needs of the research. The CFIMS has the capabilities of system-under-test (SUT) functional stimulus generation, fault injection and state monitoring, all of which are supported by a configuration capability for setting up the system as desired for a particular experiment. This report summarizes the work accomplished so far in the development of the CFIMS concept and documents the first design realization.

15. SUBJECT TERMS
Data communication systems; Distributed systems; Fault injection; Fault tolerance; System monitoring

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER ***(Include area code)***
U	U	U	UU	202	(443) 757-5802